Compaq AlphaServer SC User Guide

Quadrics Supercomputers World Ltd. **Document Version 6 - AA-RLB1A-TE**

QSW's web site can be found at

`http://www.quadrics.com/`

QSW's address in the UK is:

QSW Limited
One Bridewell Street
Bristol
BS1 2AA
UK

Tel: +44-(0)117-9075375
Fax: +44-(0)117-9075395

QSW's address in Italy is:

QSW Limited
Via Marcellina 11
00131 Rome
Italy

Tel: +39-06-4123-8615
Fax: +39-06-4191-694

Circulation Control: None

Document Revision History

Revision	Date	Author	Remarks
1	July 1999	HRA	Initial Draft
2	September 1999	HRA/DR	Final Draft
3	October 1999	DR	RMS 2.33 Release
4	October 1999	DR	RMS 2.36 Release
5	Jan 2000	DR	Additions to copyright notice
6	Apr 2000	DR	Draft changes for Product Release

Contents

List of Figures

1

Introduction

1.1 Scope of Manual

This manual describes how to use the Resource Management System (RMS). RMS provides a programming environment for running parallel programs. The manual's purpose is to provide a user's view of RMS.

1.2 Audience

This manual is for users who run applications on a Compaq AlphaServer SC system operating under RMS and for programmers who develop and run parallel programs on such a system.

The manual includes programming examples which assume that the reader is familiar with the C programming language.

1.3 Using this Manual

This manual contains four chapters and one appendix. The contents of these are as follows:

Chapter 1 (*Introduction*)

describes the layout of the manual and the conventions used to present information.

Chapter 2 (*Getting Started*)

describes how to use RMS to run a simple parallel program.

Chapter 3 (*RMS User Commands*)

introduces the RMS user commands.

Chapter 4 (*MPI and Shmem Programming*)

describes how to compile and run a parallel program.

Appendix A (*RMS Commands*)

contains manual pages for each of the RMS user commands.

Appendix B (*Shmem Library Routines*)

provides details of the implementation of Shmem.

Appendix C (*Elan Library Environment Variables*)

describes Compaq AlphaServer SC specific environment variables that can be used by MPI programs.

1.4 Related Information

The following manuals provide additional information about RMS:

- *Resource Management System Reference Manual*
- *Resource Management System Administrator's Reference Manual*

1.5 Location of Online Documentation

Online documentation in HTML format is installed in the directory `/opt/rms/docs/html` and can be accessed from a browser at `http://rmshost:8081/html/index.html`. PostScript and PDF versions of the documents are in `/opt/rms/docs`. Please consult your system administrator if you have difficulty accessing the documentation.

New versions of this and other Quadrics documentation can be found on the Quadrics web site `http://www.quadrics.com`.

1.6 Reader's Comments

If you would like to make any comments on this or any other Quadrics manual, please send them to `support@quadrics.com`.

1.7 Conventions

The following typographical conventions have been used in this document:

`monospace type`

Monospace type denotes literal text. This is used for command descriptions, file names and examples of output.

`bold monospace type`

Bold monospace type indicates text that the user enters when contrasted with on-screen computer output.

`italic monospace type`

Italic (slanted) monospace type denotes some meta-text. This is used most often in command or parameter descriptions to show where a textual value is to be substituted.

italic type

Italic (slanted) proportional type is used in the text to introduce new terms. It is also used when referring to labels on graphical elements such as buttons.

`Ctrl/x`

This symbol indicates that you hold down the `Ctrl` key while you press another key or mouse button (shown here by `x`).

TLA

Small capital letters indicate an abbreviation (see Glossary).

`ls(1)`

A cross-reference to a reference page includes the appropriate section number in parentheses.

2

Getting Started

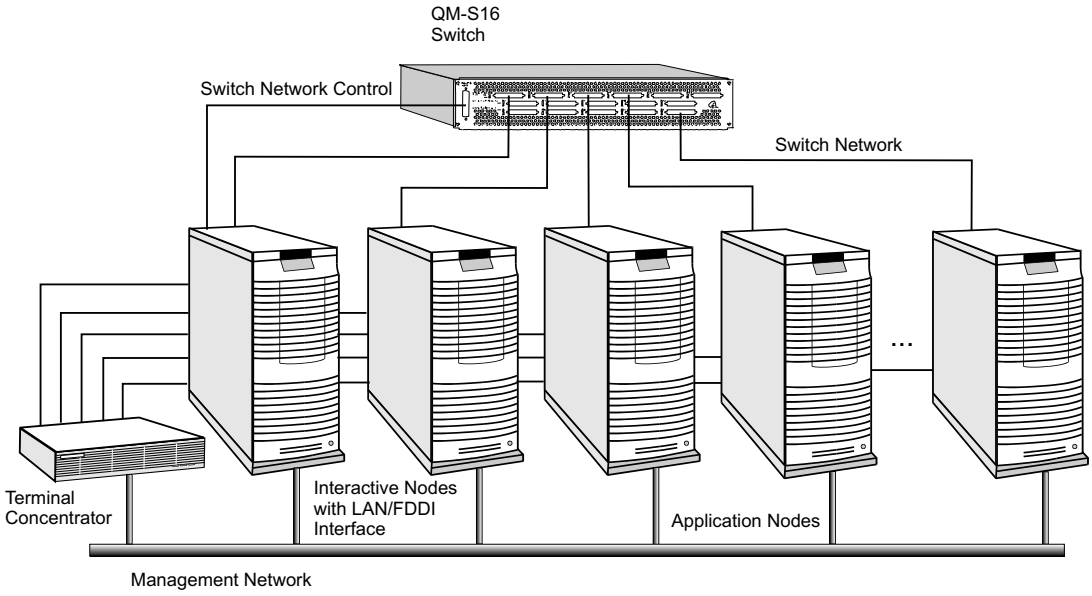
2.1 Introduction

This chapter provides an overview of the Resource Management System, the software that controls access to the resources of a Compaq AlphaServer SC system. It describes how to log into the system, get help on RMS and run a simple parallel application using the RMS services.

2.2 About RMS

The Compaq AlphaServer SC system comprises a cluster of computers (nodes) as shown in Figure 2.1.

Figure 2.1: A Cluster of Nodes



The nodes, which can be uniprocessor or multiprocessor computers, are connected by a high performance data network and a management Ethernet. Each node runs a copy of the standard UNIX® operating system.

One or more of the nodes in the cluster is used interactively. These login nodes are generally connected to an external local area network (LAN). The application nodes, used for running parallel programs, are accessed solely through RMS.

The RMS daemons, which manage the system, reside either on one of the interactive nodes or on a separate management node. This node, which runs RMS, is given the hostname alias of `rmshost`.

Nodes in an RMS cluster can be divided into *partitions*. Your system administrator may have created partitions to dedicate resources to a particular activity or group of users. The set of partitions running at any point in time is called the *active configuration*. An RMS cluster may operate in different configurations (at different times of the day), reflecting a changing pattern of resource allocation.

2.2.1 The User Interface

RMS provides a number of command line utilities that interact with the system on the users' behalf. These utilities perform tasks of general use such as querying the system's resources, loading parallel programs and running them.

Some of the utilities are specifically for system administrators. These utilities are described in the *Resource Management System Reference Manual* together with details of the RMS daemons, which manage the system. This manual concentrates on the utilities designed for users. These are as follows:

`prun`	This loads and runs parallel programs. A parallel program is a set of UNIX processes distributed over the nodes in a partition. The processes communicate over the data network using MPI or shmem libraries.
`rinfo`	This displays information about the resources available and shows which applications are running.
`rmsexec`	This runs a sequential program on a lightly loaded node.
`rmsquery`	This submits SQL queries to extract information from the RMS database. As a user, you have read access to the database.
`allocate`	This preallocates a set of resources for running a series of jobs on the same nodes.

2.2.2 The RMS Daemons

The RMS daemons manage the system, they interact via sockets and the RMS database. Each daemon is responsible for a different aspect of RMS. The daemons are largely transparent to the users of RMS. They are described here to provide users with background information on the operation of their system. For more details see the *Resource Management System Reference Manual*.

Machine Manager

The Machine Manager, also known as `mmanager`, oversees the physical operation of the *machine* — the cluster of nodes connected together by the network and running RMS.

Partition Manager

The Partition Manager, also known as `pmanager`, controls the use of nodes, the allocation of resources to users and the scheduling of parallel programs on each partition. There is a Partition Manager for each partition.

Switch Network Manager

The Switch Network Manager, also known as `swmgr`, supervises the operation of the switch network, checking for failures and isolating them.

Event Manager The Event Manager, also known as `eventmgr`, is responsible for handling events generated by the other daemons. For example it will run a handler script in response to events such as a node crash of fan failure. It also provides an interface to clients that wish to wait for certain events to occur.

Transaction Log Manager

The Transaction Log Manager, also known as `tlogmgr`, instigates changes in system state that have been requested in the *Transaction Log*. All such changes made through this mechanism to ensure that changes to the database are serialised and an audit trail is kept.

The RMS per-node Daemon

The RMS per-node daemon, also known as `rmsd` runs on each node in the system. It loads and runs user processes and monitors resource usage and performance.

2.3 Getting Started

The nodes in an RMS cluster run a standard UNIX operating system. This means they have the usual UNIX command shells, editors, compilers, linkers and libraries; they run the same applications. RMS extends standard UNIX by providing utilities for running parallel applications as well as sequential ones.

If you are not familiar with UNIX, please refer to the user documentation supplied with your system. For Sparc systems, online documentation is available at the following Web site:

For Alpha systems, see the following Web site:

`http://www.unix.digital.com/faqs/publications/pub_page/doc_list.html`

Information on the windowing system, the Common Desktop Environment, is also available on these sites. The standard textbook on UNIX is *The UNIX Programming Environment* by Kernighan and Pike. This provides a general introduction to the standard UNIX utilities and command shells.

2.3.1 Logging In

The Compaq AlphaServer SC system is generally accessed across a LAN from a workstation or terminal connected to the LAN. You log in to the system as shown here:

```
user@workstation: telnet tazmo
```

The names in this example are used as follows:

`user`	The name of the user.
`workstation`	The hostname of the workstation.
`tazmo`	The hostname of the Compaq AlphaServer SC system.

Substitute the appropriate names for your own installation. If you don't know them, ask the system administrator.

As the connection is being made, some messages may be displayed on the screen, giving the Internet address of the Compaq AlphaServer SC system and its hostname. When the connection has been established, a login prompt appears. Enter your login name and, when prompted, your password. The password is not displayed on the screen as you enter it.

```
login: user
Password:
```

When you enter the correct password, the system logs you in. A 'message of the day' may be displayed and the command prompt appears.

```
Last login: Mon Nov  2 12:22:22 from diplodocus
You have mail.
user@tazmo:
```

A word of warning about passwords: UNIX security is based on keeping passwords secret. If other people know your password then they can tamper with your work. The operating system provides a controlled mechanism for sharing work and data using access permissions; keep your password secret.

2.3.2 Setting Up Your Shell

Under Tru64 UNIX the PATH environment variable includes the directory `/usr/bin`. This is sufficient for the RMS commands but additional directories may be required for third party products. See Section 4.4 for information on the TotalView debugger and Section 4.5 for information on the Vampir visualisation tool and consult the respective user manuals for more details.

2.3.3 Getting Help

Online RMS documentation is supplied for use in these formats:

1. RMS release notes and manuals are supplied in HTML format for use with a Web browser such as Netscape Navigator or Internet Explorer.

2. RMS release notes and manuals are supplied in PDF for use with a PDF reader. They are also provided in PostScript format for printing.
3. Manual pages for the RMS commands are supplied for use with the UNIX `man(1)` command.

Using a Web Browser

For convenience, the current version of the Netscape Web browser is bundled with RMS and can be used with a local X server. Netscape can be found in `/opt/rms/bin` on the RMS host node and you start it by typing:

```
user@tazmo: netscape
```

You can use a local copy of a suitable Web browser instead. If you would like to run a local copy but don't have Netscape or Internet Explorer installed, you can get evaluation copies from `http://www.netscape.com` and `http://www.microsoft.com` respectively.

When you have started your Web browser, enter the URL for the documentation. In a standard installation, the documentation is located at the following URL:

`http://rmshost/docs/index.html`

where `rmshost` is the hostname alias of the node that runs RMS.

Using the man Program

Manual pages provide concise summaries of commands and the files that the commands use. They are useful if you already know something about the command or file and wish to find out more. You can use the `man(1)` command to provide information about itself by entering the following command:

```
user@tazmo: man man
```

At the end of each page of information, `man` pauses. Press the space bar to read the next page or enter `q` to quit (`man` uses the `more(1)` command to display its pages).

The following command displays information about the C shell:

```
user@tazmo: man csh
```

For ease of access, the manual pages for the RMS commands are included in Appendix A (*RMS Commands*) of this manual.

2.4 Running a Parallel Program

To run a parallel program, you use the RMS utility called `prun`. Without writing any code, you can experiment with `prun` right away. In this example, we use the UNIX program `uname(1)` with the options `-n`. This prints out the hostname of the workstation on which `uname` is executed.

```
user@tazmo: prun -N 4 uname -n
tazmo-0
tazmo-1
tazmo-2
tazmo-3
```

The example is a very simple parallel application in which four copies of the sequential program `uname` are executed at the same time, one per node. There is no interprocess communication.

Note that this example requires that the system administrator has set up a default partition.

You run a parallel application that does have communicating processes in the same way, using `prun` to load and execute the processes. Try running one of the example programs in the directory `/opt/rms/bin`, as follows:

```
user@tazmo: prun -N 4 dping
  2:        0 bytes      2.42 uSec     0.00 MB/s
  0:        0 bytes      2.44 uSec     0.00 MB/s
```

`prun` is the *controlling process* of your parallel program: `stdio` from the processes in the parallel program is routed back to `prun`. If `prun` is killed, the processes started by `prun` are also killed.

2.4.1 Partitions

In both of the examples shown here, `prun` executes the parallel application on the default partition. This should have been setup by your system administrator. To run a parallel program on a specific partition, use:

```
user@tazmo: prun -p small -N 2 dping
  0:        0 bytes      2.47 uSec     0.00 MB/s
```

where `small` is the name of a partition on your machine.

Each partition is controlled by a scheduler (the Partition Manager). The scheduler shares the processing resources of the partition between competing jobs. For example, a partition with ten nodes could run two five-node jobs concurrently. However, if one job

required eight nodes and another job required six, the scheduler would interleave the two jobs, giving each a certain amount of run time before suspending it so that the other one could run. This feature, called *timesharing*, may not be enabled on your system. If it is not the second job will block until the first has completed. A smaller job, requiring only 2 nodes would run. This is called *space sharing*.

The scheduling policy for a partition is controlled by its `type`. Support types are `parallel` (the partition only runs parallel jobs), `login` (partition runs UNIX® login shells and load balanced sequential tasks) `general` (all of the above) and `batch` (partition is under the exclusive control of a batch system). See *Resource Management System Reference Manual* for more information on RMS job scheduling.

2.4.2 Priorities and Projects

If one of the jobs had a higher priority than the other, the higher priority job would run to completion before the lower priority job started. In fact, if the lower priority job was already running, the scheduler would suspend it to make way for the higher priority job. Priorities are set by the system administrator. They can be assigned to groups of users (such a group is called a *project*) to give them preferential access. Priorities and projects are discussed in more detail in Section 3.5, and *Resource Management System Reference Manual*

3

RMS User Commands

3.1 Introduction

RMS provides a set of commands for running parallel programs and monitoring their execution. The set includes utilities that determine what resources are available and commands that request allocation of resources. This chapter describes how to use these tools.

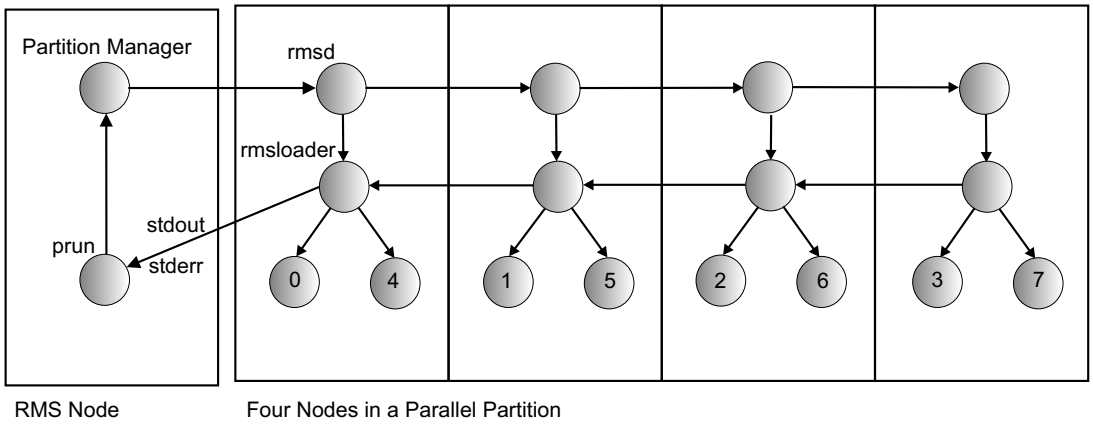
3.2 More on Parallel Programs

A parallel program consists of a controlling process (`prun`) and a set of application processes distributed over the nodes in a partition. Each application process can have multiple threads running over one or more CPUs.

RMS assigns a unique number, known as the *rank* to each process in a parallel program. The numbers range from `0` to `n-1`, where `n` is the number of processes in the program. The processes in a parallel program can communicate with each other across the data network. They do this by calling routines from one of the inter-process communication libraries passing it the rank of the remote process. RMS includes MPI and Shmem libraries that have been optimised for the Quadrics data network. These libraries are described in more detail in Chapter 4 (*MPI and Shmem Programming*).

The `prun` command sends a request to the Partition Manager to allocate the resources (CPUs and memory) required by the parallel program. Once the resources are available, the Partition Manager instructs the RMS per-node daemons to load an RMS process called `rmsloader` on each node. `rmsloader` forks and execs the application processes

Figure 3.1: Loading and Running a Parallel Program



as shown in Figure 3.1.

The `rmsloader` processes forward output printed on `stdout` and `stderr` to `prun` and hence to the terminal or output file.

The parallel program terminates when all its processes have exited. The exit status returned is formed by a global `OR` of the exit status of each process. The UNIX convention is that an exit status of zero indicates that a program has completed successfully. A non-zero exit status indicates that a problem has occurred.

If one or more of the application processes is killed by a signal (`SIGSEGV` for example) then `rmsloader` will run the corefile analysis script `/opt/rms/etc/core_analysis` which will print information on why the job failed.

3.3 RMS User Commands

The following command line utilities are described in this chapter:

- `rinfo` tells you about the resources available on your system.
- `prun` runs parallel programs.
- `allocate` allows you to preallocate resources for running a sequence of jobs on the same nodes.
- `rmsexec` starts processes on lightly loaded nodes (for example, on a node with free memory or idle CPUs).

3.4 Getting Resource Information with rinfo

Before invoking prun to run a parallel program, you can check which resources are available for your use. RMS enables the system administrator to configure the cluster of nodes into a number of different partitions, each with different numbers of nodes and different sets of resource quotas and priorities. This means that there may be restrictions on which resources you can use.

Moreover, the administrator can set up a number of these configurations and switch between them to suit different working patterns. For example, during the daytime, many users may be competing to use the resources for development purposes, whereas, at night, there may be a few large production jobs that run for a long time with no user interaction.

The command rinfo shows how the machine is configured, who is using the resources and what jobs these users are running on their resources. The following is an example of the output from rinfo:

```
user@tazmo: rinfo
MACHINE        CONFIGURATION
tazmo          day

PARTITION      CPUS    STATUS        TIME   TIMELIMIT  NODES
root              6                                   tazmo[0-2]
parallel        2/4   running    01:02:29             tazmo[0-1]

RESOURCE       CPUS    STATUS        TIME    USERNAME  NODES
parallel.996      2 allocated       00:05        user  tazmo0

JOB            CPUS    STATUS        TIME    USERNAME  NODES
parallel.1115     2   running       00:04        user  tazmo0
```

The output is split into four sections:

1. Machine

   This section shows the following:

   - The name of the machine
   - The name of the active configuration

2. Partitions

   This section has an entry for each partition in the active configuration. Each entry shows the following:

   - The partition name
   - The number of processors in use and the total.

- The status of the partition
- The partition's up time, in hours, minutes and seconds
- The upper time limit imposed on jobs, in hours, minutes and seconds
- The names of the nodes in the partition

3. Resources

   This section has an entry for each resource that has been allocated. A resource in this context means a set of processors with their associated memory and devices. Each entry shows the following:

   - The name of the resource (generated automatically)
   - The number of processors assigned to the resource
   - The status of the resource
   - The amount of time the resource has been allocated, shown in hours, minutes and seconds
   - The name of the user who has been allocated the resource
   - The names of the nodes that provide the resource

4. Jobs

   This section has an entry for each job running on the machine. Each entry shows the following:

   - The name of the job (generated automatically)
   - The number of processors that the job is using
   - The status of the job
   - The amount of time the job has been running, shown in hours, minutes and seconds
   - The name of the user who is running the job
   - The names of the nodes across which the job is distributed

   If no resources are in use, only the machine and partition sections are displayed.

3.4.1 Specifying Node Names

Note that node names are specified by a pattern-matching syntax used by the UNIX shell (see `glob(3)`). The numbers in square brackets all share the common stem that precedes the square brackets. Within the square brackets, two numbers separated by a hyphen denote an inclusive range while numbers separated by commas or white space represent a list. The UNIX pattern-matching syntax is extended to include numbers of more than one digit. For example, `tazmo[8-10,12,15]` refers to the nodes named: `tazmo8`, `tazmo9`, `tazmo10`, `tazmo12` and `tazmo15`.

3.4.2 Command Line Options

rinfo has a number of command line options that let you restrict or expand the amount of information displayed. See Appendix A (*RMS Commands*) for more details. In this chapter, we examine some of the more commonly used options.

```
rinfo [-achjlmnpqr] [-L [partition] [statistic]] [-s daemon [hostname]]
    [-t node|name]
```

You can use the -h option with all the RMS commands to get a list of the available options.

3.4.3 Querying the Machine's Users

Specify the -a option to list the resources and jobs of all users, as shown in the following example:

```
user@tazmo: rinfo -a
PARTITION      CPUS    STATUS        TIME   TIMELIMIT  NODES
root              6                                   tazmo[0-2]
parallel        4/4   running    01:02:29             tazmo[0-1]

RESOURCE       CPUS    STATUS        TIME    USERNAME  NODES
parallel.996      2 allocated       00:05        user  tazmo0
parallel.997      2 allocated       00:02        dave  tazmo0

JOB            CPUS    STATUS        TIME    USERNAME  NODES
parallel.1115     2   running       00:04        user  tazmo0
parallel.1116     2   running       00:02        dave  tazmo0
```

To restrict the rinfo output to the jobs section only, specify the -j option. Specify rinfo with both the -a option and -j option to display the jobs section for all users. To display your own jobs only, omit the -a option, as shown in following example:

```
user@tazmo: rinfo -j
JOB            CPUS    STATUS        TIME    USERNAME  NODES
parallel.1115     2   running       00:04        user  tazmo0
```

3.4.4 Checking Quotas

The system administrator can set limits on the way a partition is used. Usually, these limits are set at the project level so that each user working on a project is automatically subject to the limits set for the project as a whole. For example, there might be a limit of 20 on the number of processors available for project alpha at any one time. Therefore, if two users, working on project alpha, each had 8 processors allocated to them, only 4

processors would be available for a third project member, even though rinfo might show that the partition had 64 processors and there were no other users.

The system administrator can limit the following values.

- The maximum number of CPUs.
- The maximum amount of memory per process,
- The scheduling priority,

You can check what restrictions have been set by running rinfo with the -q option.

```
duncan@gold0: rinfo -q
PARTITION       CLASS         NAME     CPUS   MEMLIMIT
parallel         user       duncan      0/8        256
parallel      project        alpha    16/20        256
parallel      project      default     4/16        256
```

The system administrator can also set limits on individual users. These are always more restrictive than those imposed at the project level.

Users can request lower values of each of these values. You can specify a per-process memory limit by setting the environment variable RMS_MEMLIMIT before using prun. This may help in getting their jobs run sooner.

Users can request that their jobs be assigned a priority lower than the default by setting the environment variable RMS_PRIORITY before using prun.

You can specify the name of the project you are working on by setting the environment variable RMS_PROJECT before using prun. All the environment variables that can be used with RMS are described in Appendix A (*RMS Commands*).

3.4.5 Viewing Configuration Details

The -c option displays the names of all the machine configurations:

```
user@tazmo: rinfo -c
day
night
```

You can find out the names of all the active partitions with the -p option. This also gives the number of CPUs in each partition:

```
user@tazmo: rinfo -p
parallel 16
root      2
login     4
```

The -m option displays the name of the machine:

```
user@tazmo: rinfo -m
tazmo
```

3.5 Running Programs with prun

prun is the RMS utility for running parallel programs. prun loads multiple copies of a single application program onto a range of nodes and runs them. prun acts as the program's interface to RMS, handling stdio and forwarding certain signals.

You specify to prun how many processes to load and on which partition. In addition, the prun options (see Appendix A (*RMS Commands*)) enable you to select more precisely the distribution of the processes in the partition. Unless you have already allocated a resource for the program (see Section 3.6), prun does so on your behalf, blocking until a appropriate CPUs becomes available.

3.5.1 Command Line Options

The syntax of the prun command is as follows:

```
prun [-hiOstv] [-B basenode] [-c cpus] [-n procs] [-N nodes]
    [-m block|cyclic] [-p partition] program [args ...]
```

You can use the -h option to get a list of the available options and valid arguments. See Appendix A (*RMS Commands*) for full details.

3.5.2 Selecting a Partition

With the -p option, you can select a partition for the program. If you omit this option, a default partition, nominated by the system administrator, is used.

In the following example, two copies of myprog are loaded onto the partition called parallel:

```
user@tazmo: prun -p parallel -n 2 myprog
Hello from myprog
Hello from myprog
```

You can specify your own default partition by setting the environment variable RMS_PARTITION. If no partition has been specified and the system administrator has not set up a default, you will get an error message.

3.5.3 Specifying Processes and Nodes

You can specify how many instances of a program to run by using the -n option as shown in the previous section. You can also specify how the processes are distributed across the nodes in the partition. In the following example, two instances of uname are requested. All arguments after the program name are passed to each process.

```
user@tazmo: prun -n 2 uname -n
tazmo4
tazmo4
```

Note that both instances ran on the same node. By default, RMS allocates one process per processor, using all the processors on one node before moving on to the next. You can override this behaviour; by using the -N option, which specifies how many nodes are required for the program. In the following example, four nodes are requested and an instance of uname is executed on each:

```
user@tazmo: prun -N 4 uname -n
tazmo4
tazmo5
tazmo6
tazmo7
```

The -n and -N options can be used in combination to place more than one process on each node. In the following example, four processes are executed, two per node:

```
user@tazmo: prun -n 4 -N 2 uname -n
tazmo4
tazmo4
tazmo5
tazmo5
```

The RMS scheduler will allocate one CPU per process, dividing them evenly over the requested number of nodes (provided n is divisible by N). If you are not concerned with how the processes in your application are distributed over nodes then use the -n option alone and your application will be run as soon as CPUs are available. If you require the same number of CPUs on each node and a contiguous range of nodes then use the -N option. The -c option allows you can select how many CPUs you want for each process. This is for use in multi-threaded applications.

The following example runs four myprog processes on two nodes, allocating two CPUs per process – 8 CPUs in total:

```
user@tazmo: prun -c 2 -n 4 -N 2 myprog
```

RMS does not take any stance on how the additional CPUs are to be used. This is up to the program.

RMS does not normally run more processes per node than there are CPUs. However, there are circumstances in which this can be useful. The -O allows you to do this.

```
user@plague0: prun -n5 -N1 hostname
prun: Error: can't allocate 5 cpus on 1 node: max cpus per node is 4

duncan@plaguei: prun -O -n5 -N1 hostname
plague0.quadrics.com
plague0.quadrics.com
plague0.quadrics.com
plague0.quadrics.com
plague0.quadrics.com
```

The -m option to prun allows you to control how the processes are distributed over nodes. Options are block (the default) and cyclic. This is illustrated below together with the -t option that prefixes the process number onto each line of output.

```
duncan@plaguei: prun -n4 -N2 -t -mblock hostname
0 plague0
1 plague0
2 plague1
3 plague1
duncan@plaguei: prun -n4 -N2 -t -mcyclic hostname
0 plague0
2 plague0
1 plague1
3 plague1
```

3.5.4 Input and Output

Each process in a user's application has three standard input/output (I/O) streams:

1. stdin or unit 5 in Fortran
2. stdout or unit 6 in Fortran
3. stderr or unit 0 in Fortran

The use of these streams by parallel programs is different from that of sequential programs (that is, standard UNIX applications that execute independently of all other processes).

When the parallel processes start executing, stdout and stderr are routed to prun. Normal write operations to these file descriptors have the expected effect, as do calls to the isatty(3c) function. Other ioctl functions are not reliable.

In a parallel program the three I/O streams should be used as follows:

stdin This must be redirected to come from a file.

stdout	This is used for line buffered output from all processes.
stderr	This is used for unbuffered output from all processes.

Getting Input

Processes executed by prun cannot read from stdin. This is because repeatable behavior cannot be guaranteed when unsynchronized processes read at the same time. You can work around this by running a shell script that executes the program with stdin redirected from a file. In the following example, with the first command all processes read from the same file; with the second, the processes have a file each:

```
user@tazmo: prun sh -c 'myprog < myfile'
user@tazmo: prun sh -c 'myprog < tmp.$RMS_RANK'
```

The -c option to the shell (the Bourne Shell, in this example) tells it that the next string represents a command to execute. Single quotes are used to delimit the string, not double quotes. This method of quoting prevents the shell from trying to expand $RMS_RANK before the command is executed. RMS_RANK is set by prun to the rank of the process. Similarly, each process can direct its output to a unique file:

```
user@tazmo: prun sh -c 'uname -n > host.$RMS_RANK'
```

The following simple shell script runs the first process in an xterm window with stdin, stdout and stderr for that process redirected to the new window:

```
#!/bin/sh~
if [ $RMS_RANK -eq 0 ] ; then
  xterm -e myprog
else
  myprog
fi
```

Line Buffered Output

In a parallel program, when processes print simultaneously to stdout, the output comes out on separate lines but in an arbitrary order. In fact, the ordering may be different each time the program runs. The -t option of prun tags the output of each process with the rank of the process. This makes it easy to identify the source of messages output by the program, as shown in the following example:

```
user@tazmo: prun -n 4- t pwd
2 /home/user
0 /home/user
3 /home/user
1 /home/user
```

3.5.5 RMS Environment Variables

The RMS environment variables are described in full in Appendix A (*RMS Commands*). We have already mentioned two: one that prun sets, the variable RMS_RANK; and one that prun reads, the variable RMS_PROJECT. Another environment variable that you can set is RMS_IMMEDIATE. This tells prun to exit rather than block if the resources required for the program are not available immediately.

The following example sets two environment variables (from the C shell) and uses some environment variables created by prun:

```
user@tazmo: setenv RMS_IMMEDIATE
user@tazmo: setenv RMS_PROJECT database
user@tazmo: prun -n 4 csh -c 'echo process $RMS_RANK of $RMS_NPROCS'
process 3 of 4
process 2 of 4
process 0 of 4
process 1 of 4
```

First of all, RMS_IMMEDIATE is set so that prun will not block if insufficient resources are available. Instead, it will return immediately. This has the same effect as running prun with the -i option.

Then we specify with RMS_PROJECT that the subsequent jobs belong to the project called database. This will affect the CPU usage limit applied to the jobs and also the accounting records, which RMS keeps. If you do not set RMS_PROJECT you get the default values set by the system administrator.

When prun runs the four processes as requested, the rank of each instance of echo is displayed together with the total number of processes, RMS_NPROCS. RMS passes all the environment variables to the processes it executes.

3.5.6 Memory Limits

RMS can impose memory limits on the processes in a parallel application. The default limits ensure that each each process has a fair share of the memory available. The system administrator can raise or lower the limits on a per user or per project basis.

Use rinfo with the -q option or prun with the -v option to see the limits that apply to you.

```
duncan@cfs1: rinfo -q
PARTITION       CLASS         NAME     CPUS   MEMLIMIT
parallel      project      default     0/22         96

duncan@cfs1: prun -v -n2 dping
prun: starting 2 processes on 2 cpus memlimit 96 MB no timelimit
  0:        0 bytes      2.43 uSec     0.00 MB/s
```

Exceeding these limits will cause brk(), malloc(), mmap() and sbrk() system calls to fail with ENOMEM. Programs with static arrays larger than the memory limit will fail immediately. For example if you run a process like this with too small a memory limit it will exit before entering main and prun will exit immediately.

```
#include <stdio.h>

int array[50 * 1024 * 1024];

main (int argc, char **argv)
{
    printf("hello world\n");
    exit(0);
}
```

If you use prun with -v it will print a warning if all processes exit this way

```
duncan@cfs1: prun -v -n2 myprog
prun: starting 2 processes on 2 cpus memlimit 10 MB no timelimit
prun: Warning: exit 1 on all nodes. Data segment size may exceed memory limit.
```

You can check on the size of your application processes with the command size

```
duncan@cfs1: size myprog
text    data    bss     dec     hex
8192    8192    209707392       209723776       c802180
```

In this case you will need 200 MBytes per process. To set this limit use the enviornment variable RMS_MEMLIMIT

```
duncan@cfs1: setenv RMD_MEMLIMIT 200
```

before starting your program. The units are MBytes per process.

3.5.7 Program Termination

A parallel program terminates when all its processes have exited or when one or more processes is killed by a signal. If a program exits cleanly the exit status returned is formed by a global OR of the exit status of each process. This allows an application to return a small number of carefully chosen non-zero exit status values when something goes wrong.

If one or more of the application processes is killed (for example by the signal SIGSEGV) prun will exit immediately with a status value indicating the signal number.

```
$ prun -v program
...
```

```
prun: program (pid 767544) killed by signal 11
...
$ echo $?
139
```

The exit status is 128 plus the signal number that caused the process to be killed.

When RMS detects that a process it started was killed by a signal it will run a corefile analysis script. This script looks for core files created by your program and prints out a backtrace showing you where the process failed (see Section 3.5.8 for details).

If the processes in a parallel program are started using a shell script then by convention the shell script will exit with a status of 128 plus the signal number. RMS interprets this as an error and the program is terminated.

3.5.8 Corefiles

The rms policy on handling core files when a process within a parallel job fails is as follows:

- Kill off remaining processing in the job - they are useless anyway.
- Generate core in /local/core/rms/<id>/<extended-core-filename>
- Run the debugger on the core file and put the backtrace to stderr
- Delete the core file and directory. The deletion happens when rms is freeing the allocated resource.

The <extended-core-filename> is of the form core.<program>.<hostname>.<instance>.

The reason for deleting the core file and directory is that typically production jobs are compiled with optimizations so there is little diagnostic information available in the core file. In addition, having hundreds of useless core files scattered over the local disks will soon become a maintenance problem.

To diagnose a failing program a developer is advised to:

- Compile the program with -g for debug and symbolic information inclusion.
- Run the job by first allocating a resource, using the allocate command, and then the prun command.
- When the program fails it will produce a core file. The prun command prints its pathname.
- Copy the core file to your home directory and exit the allocate subshell.

If users want to catch core files from production runs, i.e. without allocate, then they can run the job in a script that copies the core file to a permanent lcoation or persuade their system administrator to do this in site specific core file analysis script.

```
user@tazmo: allocate -N4
user@tazmo: prun myprog
myprog: process 0 killed by signal 11
...
...
user@tazmo: rinfo -r
parallel.397
user@tazmo: prun -B0 -N1 cp '/local/core/rms/397/core*' .
user@tazmo: exit
```

3.5.9 Common Problems

There are some common problems and error messages that you may encounter when running applications. This section suggests some solutions.

The Program Hangs

The program may hang if `prun` cannot allocate the resources required for the program and has blocked, waiting until they become available. Resource requests made by `allocate` will behave in the same way.

If you enable verbose reporting with the `-v` option or by setting `RMS_VERBOSE`, `prun` will output a message as your jobs starts

```
duncan@gold0: prun -v -N2 uname -a
prun: starting 2 processes on 2 cpus memlimit 96
OSF1 gold0.quadrics.com T5.0 861.3 alpha
OSF1 gold1.quadrics.com T5.0 861.3 alpha
```

If the job is blocked waiting for resources you will get a warning message. Some time later you will get the start message and then the output from your program.

```
duncan@gold0: prun -v -n2 uname -a
prun: Warning: waiting for free cpus
prun: starting 2 processes on 2 cpus memlimit 96
OSF1 gold0.quadrics.com T5.0 861.3 alpha
OSF1 gold0.quadrics.com T5.0 861.3 alpha
```

If `prun` hangs between the two messages, you can suspend it by pressing `Ctrl/Z` and use `rinfo` to find out what is going on. You should see that your resource request is `queued` or `blocked`. If the request is `blocked` it must wait for the completion of other jobs that you or your project have submitted. If the request is `queued` it must wait for the completion of jobs submitted by other users or projects.

You can prevent your job from blocking with the -i option to prun or by setting the environment variable RMS_IMMEDIATE. This will cause the job to fail if resources are not available.

Error Messages

The error messages you may encounter are as follows.

`prun: can't find program`

The problem here is be that the specified program cannot be located using your current search path. The solution is to add the program's directory to your PATH environment variable.

`prun: Error: Partition manager for partition is down`

The problem in this case is that the partition you specified with the -p option (or the default partition, if you did not specify one) is unavailable. The solution is to specify an alternative partition or ask the system administrator to restart the partition. rinfo (see Section 3.4 or Appendix A (*RMS Commands*)) will tell you the status of the various partitions.

`prun: Error: no such partition as` *name*

The problem here is clear: the specified partition does not exist. The most likely cause of this error is an incorrectly entered partition name. Once again, rinfo will show you the names of the partitions.

`prun: failed to start loader`

The per node daemon, rmsd, starts an rmsloader process on each node. These loaders connect back to prun, as shown in Figure 3.1. They find out from prun what they should run and then exec(3) the user's program and forward its I/O. This should all happen quickly. Things can go wrong, however, so a timeout is built into prun. This is the message that appears when the timeout expires.

Your job is killed

If your job is killed prun will exit early with a signal status of 137 (128 + SIGKILL) and a message

```
$ prun -v dping -n1000000 0 8k
prun: starting 2 processes on 2 cpus default memlimit no timelimit
  0:        0 bytes      2.69 uSec     0.00 MB/s
  0:        1 bytes      3.54 uSec     0.28 MB/s
```

```
  0:        2 bytes      3.70 uSec     0.54 MB/s
prun: loaders exited without returning status
$ echo $?
137
```

Simillar messages are printed when a node that the job was running on crashes.

3.6 Allocating Resources with `allocate`

RMS makes a distinction between allocating a resource (CPUs and memory) and running jobs on it (see Section 3.4 for more details on resources). `prun` combines both tasks: allocating resources and running jobs. `allocate` simply allocates resources.

You may find it useful to use `allocate` before `prun` if you want to run a sequence of jobs with the same resource requirements. This means you only have to wait once for the CPUs to be allocated. It is also useful if you want to run several jobs concurrently.

There are two ways to use `allocate`. These are described in the next two sections. Basically, `allocate` has an optional and final argument which is the name of a shell script. If you do not specify a script, `allocate` spawns an interactive shell that has the resource until you exit the shell. If you do specify a shell script, the resource is allocated to this script until it exits.

3.6.1 Command Line Options

`allocate` has four options that have the same option letter and meaning as their `prun` counterparts. See Appendix A (*RMS Commands*) for full details.

```
allocate [-hiv] [-Bbase] [-Ccpus] [-Nnodes] [-ppartition] [script [args ...]]
```

The most frequently used options to `allocate` are `-N` which allows you to specify the number of nodes and `-C` the number of CPUs per node. The `-N` option takes either a numeric argument specifying the number of nodes to be allocated. Alternatively you can use the argument `all` to allocate all nodes in the partition.

As with most RMS commands you can use the `-h` option to get a list of the available options and valid arguments.

3.6.2 Allocating Resources to an Interactive Shell

When `allocate` is run without specifying a shell script as the final argument, it spawns an interactive shell that has the resources allocated to it. These resources are freed when you exit the shell or when a time limit, imposed by the system administrator on parallel jobs, expires; whichever comes first. In the following example, both the `prun` commands execute concurrently on the partition called `parallel`:

```
user@tazmo: allocate -N 4 -p parallel
user@tazmo: prun -n 2 myprog &
user@tazmo: prun -n 2 test
user@tazmo: exit
```

The & following the program name and associated options tells the shell to run the program in the background and return to the command prompt.

In the next example the two prun commands are executed sequentially, both on the same two nodes in the parallel partition:

```
user@tazmo: allocate -N 2 -p parallel
user@tazmo: prun uname -n
tazmo-32
tazmo-33
user@tazmo: prun uname -n
tazmo-32
tazmo-33
user@tazmo: exit
```

If this example was run without allocating resources to the shell, you could not guarantee that the second use of prun would start immediately after the first completed; nor could you guarantee that both runs would use the same two nodes in the partition.

3.6.3 Verifying Resource Allocation to a Shell

When you allocate a resource to an interactive command shell, you can check that the resource has been successfully allocated by using rinfo (see Section 3.4 and Appendix A (*RMS Commands*)) as shown in the following example:

```
user@tazmo: rinfo
MACHINE        CONFIGURATION
tazmo          day

PARTITION      CPUS    STATUS        TIME   TIMELIMIT  NODES
root              6                                   tazmo[0-2]
parallel        2/4   running    01:02:29             tazmo[0-1]

RESOURCE       CPUS    STATUS        TIME    USERNAME  NODES
parallel.996      2 allocated       00:05        user  tazmo0

JOB            CPUS    STATUS        TIME    USERNAME  NODES
parallel.1115     2   running       00:04        user  tazmo0
```

This shows that a resource named parallel.996 has been allocated to user and has been in use for 5 seconds.

Another way of verifying that the resource has been allocated to the shell is to change the shell prompt so that it shows the name of the resource. You can do this by editing

the setup file that the shell reads in when you login. As mentioned in Chapter 2 (*Getting Started*), the name and syntax of the setup file varies according to the shell you use. The edits for a C shell and Bourne shell setup file are as follows.

Changing the Prompt with the C Shell

Add the following commands to your `.cshrc` file to make the shell prompt change whenever you have been allocated resources:

```
# additions to .cshrc
if ($RMS_RESOURCEID) then
  set prompt="${RMS_RESOURCEID}:"
else
  set prompt="$user@‘uname -n‘:"
endif
```

When you next login and run `allocate` to get an interactive shell with a resource allocated to it, the prompt will be the name of the resource (as shown by `rinfo`):

```
user@tazmo: allocate -p parallel -n 6
parallel.4: exit
user@tazmo:
```

Changing the Prompt with the Bourne Shell

Add these lines to your `.profile` file to make the shell prompt change whenever you have been allocated resources:

```
# additions to .profile
if [ $RMS_RESOURCEID ] ; then
  PS1="$RMS_RESOURCEID:"
else
  PS1="$user@‘uname -n‘:" "
fi
```

When you next login and run `allocate` to get an interactive shell with a resource allocated to it, the prompt will be the name of the resource (as shown by `rinfo`):

```
user@tazmo: allocate -p parallel -n 6
parallel.5: exit
user@tazmo:
```

3.6.4 Allocating Resources to a Shell Script

Rather than allocate a resource to an interactive shell, as just described, you can allocate it to a shell script. Calls to `prun` (see Section 3.5 and

Appendix A (*RMS Commands*)) within this shell script result in the execution of parallel programs on the allocated resource.

In the following example, the shell script called `script` uses `prun` to run four user jobs, one after the other:

```
user@tazmo: cat script
#!/bin/sh
# script to run
prun preprocess
prun iterate -f12000
prun iterate -f24000
prun postprocess
```

`allocate` is used to get a resource, which comprises 8 nodes on the `parallel` partition, to give that resource to the shell script and to run the script:

```
user@tazmo: allocate -N 8 -p parallel script
```

3.7 Running a Sequential Program with `rmsexec`

`rmsexec` provides a mechanism for running sequential processes on lightly loaded nodes — nodes, for example, with free memory or low CPU usage. Note that this load balancing service may not be available on all partitions. It is a configuration option, selected by the system administrator.

3.7.1 Command Line Options

`rmsexec` has a number of options that enable you to influence the choice of node. See Appendix A (*RMS Commands*) for full details.

```
rmsexec [-hv] [-p partition] [-s stat] [hostname] program [args ...]
```

You can use the `-h` option to get a list of the available options and valid arguments.

3.7.2 Selecting a Node

`rmsexec` restricts its search to the partitions you are entitled to use (as defined by the system administrator). You can restrict the search still further by specifying a particular partition with the `-p` option, as shown in the following example:

```
user@tazmo: rmsexec -p parallel myseqprog
```

You can also request a processor on a specific node. The following example requests the node `tazmo2`:

```
user@tazmo: rmsexec tazmo2 myseqprog
```

3.7.3 Defining Load

You can specify the criterion for judging load with the `-s` option. There are four statistics that can be applied.

`usercpu`	The percentage CPU time spent in the user state.
`syscpu`	The percentage CPU time spent in the system state - a measure of the I/O load on a node.
`idlecpu`	The percentage CPU time spent in the idle state.
`freemem`	The free memory in MBytes.
`users`	Number of users.

By default, the `usercpu` statistic is used. Statistics can be used on their own in which case a node is chosen that is lightly loaded according to this statistic or you can specify a threshold. Some examples that might be of interest follow.

```
user@tazmo: rmsexec -s usercpu myseqprog
user@tazmo: rmsexec -s"usercpu < 50" myseqprog
user@tazmo: rmsexec -s"freemem > 256" myseqprog
```

4

MPI and Shmem Programming

4.1 Introduction

Programs may be run in parallel by using either the Message Passing Interface (MPI) library or the Shmem library for process synchronization and communications. This chapter introduces you to MPI and Shmem programming on the Compaq AlphaServer SC, demonstrating how to compile and link programs and run them under RMS. It also describes how to run programs under the TotalView debugger and the Vampir visualization and analysis tool. Two example programs are provided. Both show a simple `ping` application (for measuring interprocess communication latency and bandwidth): one in each programming style.

The information in this chapter is organized as follows:

- MPI overview (Section 4.2)
- Shmem overview (Section 4.3)
- Using TotalView™ to debug MPI programs (Section 4.4)
- Using Vampir to analyze MPI programs (Section 4.5)
- Using the MPI library to implement the example program (Section 4.6)
- Using the Shmem library to implement the example program (Section 4.7)

4.2 MPI Overview

This section provides an overview of the MPI library. The information is organized as follows:

- Introduction to MPI (Section 4.2.1)
- Compiling, linking and running MPI programs (Section 4.2.2)
- Further sources of information on MPI (Section 4.2.3)

4.2.1 Introduction to MPI

The MPI library is a standard message passing library for parallel applications. Using MPI, parallel processes cooperate to perform their task by passing messages to each other. MPI includes point-to-point message passing and collective operations between a user-defined group of processes.

Processes identify each other according to their *rank* in the group. The rank is an integer in the range `0` to `n-1`, where `n` is the total number of processes in the program. A process can query its rank and the size of its group.

The initial group of processes includes all the processes in the program and is known as the world group. The world group may be subdivided into subgroups. Processes can be identified according to their rank in the subgroup. In this way, virtual topologies can be created, such as graphs, which map directly onto the application domain.

A communicator is a higher-level grouping construct that contains a group and a communications context (scoping information).

Each message-passing routine has four variables that can be used to synchronize the sender and receiver: the sender's rank, the receiver's rank, a user-defined tag and the communications context. The following example shows a send routine. The sender's rank is implicit.

```
MPI_Send(txbuf, nob, MPI_BYTE, receiver, tag, MPI_COMM_WORLD);
```

`MPI_Comm_World` is a communicator that contains all the processes in the world group. This communicator is set up for the process when it is initialized for MPI. `MPI_BYTE` specifies the datatype of the message data. MPI performs data conversion transparently and supports both built-in and user-defined datatypes.

The receive routine includes a status argument, used to determine the success of the operation. Wildcards can be used for the sender's rank and the tag.

```
MPI_Recv(rxbuf, nob, MPI_BYTE, sender, tag, MPI_COMM_WORLD, &status);
```

The message-passing routines support the following:

- Blocking (synchronous), point-to-point send and receive
- Non-blocking (asynchronous), point-to-point send and receive
- Collective message-passing operations derived from the four primitives: broadcast, scatter, gather and reduce

In addition to the communications routines, MPI provides the following categories of service:

- Environmental queries
- Timing information for measuring performance
- Profiling information for monitoring performance

The MPI library is layered on top of the tagged message passing routines provided by the Elan library. These routines make use of tagged message ports, known as *tport*s, for point-to-point communications.

On a SMP node, the tagged message passing routines (and, hence, MPI) use shared memory segments to communicate between processes on the same node and the Compaq AlphaServer SC data network to communicate between nodes. There are a number of parameters that you can tune by setting environment variables to help to optimize the performance of your MPI programs. For further details see
Appendix C (*Elan Library Environment Variables*)

4.2.2 Compiling, Linking and Running MPI Programs

This section describes how to compile, link, and run MPI programs written in C and Fortran.

Compiling and Linking C Programs

To build MPI programs, use the MPI header file in your source files and specify the MPI compiler in your `make` files, as follows:

1. Include the MPI header file, `mpi.h`, in your program, with the following include directive:

   ```
   #include <mpi.h>
   ```

2. Compile the program linking it with the MPI and Elan libraries.

   ```
   cc -o myprog myprog.c -lmpi -lelan
   ```

Compiling and Linking Fortran Programs

To build MPI programs, use the MPI header file in your source files and specify the MPI compiler in your `make` files, as follows:

1. Include the MPI header file, `mpif.h`, in your program with the following include directive:

   ```
   INCLUDE 'mpif.h'
   ```

2. Compile the program with `f77` or `f90` and link with the MPI and Elan libraries.

   ```
   f90 -o myprog myprog.f -lfmpi -lmpi -lelan
   ```

Running MPI Programs

To execute an MPI program on the Compaq AlphaServer SC, enter the RMS command `prun` follwed by the name of the program:

```
user@tazmo: prun -n 4 myprog
```

The `-n` flag instructs RMS to start four copies of `myprog`. For more information on `prun` see Section 3.5 and Appendix A (*RMS Commands*).

4.2.3 Further Information on MPI

You can find more details about the MPI library from the following Web site:

`http://www.mcs.anl.gov/mpi/index.html`

Section 4.4 describes how to use TotalView to debug MPI programs on Compaq AlphaServer SC systems. Section 4.5 describes how to use Vampir to trace and analyze MPI programs on Compaq AlphaServer SC systems.

4.3 Shmem Overview

This section provides an overview of the Shmem library. The information is organized as follows:

- Introduction to Shmem (Section 4.3.1)
- Compiling, linking and running Shmem programs (Section 4.3.2)
- Further sources of information on Shmem (Section 4.3.3)

4.3.1 Introduction to Shmem

The Shmem library provides direct access (via `put` and `get` calls) to the memory of remote processes. A message passing library, such as MPI, requires that the remote process issue a receive to complete the transmission of each message; the Shmem library, by contrast, provides the initiating process with direct access to the target memory. The one-sided communication used by Shmem maps well onto the DMA hardware in the Compaq AlphaServer SC network adapter. A consequence of this is that Shmem latencies are very low.

Shmem provides the following categories of routine:

- Put routines write data to another process.
- Get routines read data from another process.
- Collective routines distribute work across a set of processes.
- Atomic routines perform an atomic fetch-and-operate, such as fetch-and-increment or swap.
- Synchronization routines order the actions of processes For instance, the `barrier` routine might be used to prevent one process from accessing a data location before another process has updated that location. The Shmem programming model requires that you think about the synchronization points in your application and the communication that must go on between them.
- Reduction routines reduce an array to a scalar value by performing a cumulative operation on some or all of the array elements. For example, a summation is a reduction that adds all the elements of an array together to yield one number.

The Shmem library also includes a number of initialization and management routines. See Appendix B (*Shmem Library Routines*)) for further information on the Shmem routines supported.

Shmem routines provide high performance by minimizing the overhead associated with data passing requests, maximizing bandwidth and minimizing data latency (the time from when a process requests data to when it can use the data). By performing a direct memory-to-memory copy, Shmem typically takes less steps to perform an operation than a message-passing system. For example, in a generic message passing system, for a put operation the sender performs a send, then the receiver performs a receive; for a get operation, the requesting process sends a description of the data required, the sender acts on the request by sending the data, then the requesting process receives the data. By contrast, Shmem requires only one step: either send the data or get the data. However, additional synchronization steps are almost always required when using

Shmem. For example, the programmer must ensure that the receiving process does not try to use the data before it arrives.

4.3.2 Compiling, Linking and Running Shmem Programs

This section describes how to compile, link, and run Shmem programs written in C and Fortran.

Compiling and Linking C Shmem Programs

To compile C Shmem programs, use the Shmem header file and library in your source and `make` files, as follows:

1. Include the Shmem header file, `<shmem.h>`, in your program, with the following include directive:

   ```
   #include <shmem.h>
   ```

2. Specify the `shmem` library to the linker with the `-l` option on the command line:

   ```
   cc -o myprog myprog.c -lshmem -lelan
   ```

Compiling and Linking Fortran Shmem Programs

To compile Fortran Shmem programs, use the Shmem header file and library in your source and `make` files, as follows:

1. Include the Shmem header file, `shmem.fh`, in your program with the following include directive:

   ```
   INCLUDE 'shmem.fh'
   ```

2. Specify the `shmem` library to the linker with the `-l` option on the command line:

   ```
   f77 -o myprog myprog.f -lshmem -lelan
   ```

Running Shmem Programs

To execute a Shmem program, enter the RMS command `prun` follwed by the name of the program:

```
user@tazmo: prun -n 4 myprog
```

The `-n` flag instructs RMS to start four copies of `myprog`. For more information on `prun` see Section 3.5 and Appendix A (*RMS Commands*).

4.3.3 Further Information on Shmem

For more information about Shmem, see `intro_shmem(3)` and the following documents:

- *CRAY T3E C and C++ Optimization Guide*, reference number: SG-2178
- *CRAY T3E Fortran Optimization Guide*, reference number: SG-2518 3.0
- Shmem reference pages

These documents are available at the following Web site:

`http://techpubs.sgi.com/`

4.4 Using TotalView

TotalView is the source-level debugger for Compaq AlphaServer SC systems. TotalView is licensed from Etnus Inc. Their web site is `http://www.etnus.com`

Version 3.9 of TotalView has been integrated with RMS and the Compaq AlphaServer SC MPI library. TotalView is has an easy-to-use interface (based on the X Window System) and support for debugging parallel programs. TotalView runs on the same node as you run `prun`, it starts a remote server process called the TotalView Debugger Server, `tvdsvr`, on each of the nodes used by you parallel program.

TotalView allows you to select which of your processes to inspect. Each is displayed in its own window together with the source code, status, program counter, threads, breakpoints, stack trace and stack frame.

TotalView cooperates with RMS to perform the following functions:

- Acquire the processes spawned by `prun` at startup, before they have entered the main program.
- Attach to a parallel job started by `prun` and acquire all of the processes in the job, wherever they reside in the machine.

In addition, using TotalView, you can attach to processes that are already running. This means that you can debug processes that were not started under TotalView control.

Before using TotalView, update your `PATH` environment variable, as described in Chapter 3 of the *TotalView User's Guide*.

4.4.1 Running a Parallel Job under TotalView Control

This section describes how to run a parallel job under the control of TotalView.

1. To start a parallel job under the control of TotalView, use the following command:

   ```
   user@tazmo: totalview prun -a prun_arguments
   ```

   where `prun_arguments` are the command line arguments for `prun`, as in the following example:

   ```
   user@tazmo: totalview prun -a -n 2 myprog 0 1
   ```

   The `-a` option is a TotalView option which specifies that the arguments which follow are for the program TotalView is running. The program is specified by the first argument to TotalView. For information about `prun`, see Section 3.5 and Appendix A (*RMS Commands*).

2. Select the following TotalView option:

   ```
   Go/Halt/Step/Next/Hold -> Go Process (g)
   ```

   When `prun` has acquired the resources to execute the job, TotalView starts remote servers on the appropriate nodes by using its remote server startup mechanism (see *Starting the Debugger Server for Remote Debugging* in Chapter 4 of the *TotalView User's Guide*).

   After the remote servers have started, TotalView acquires the processes that make up the parallel job.

3. TotalView prompts you to indicate whether you want to stop the processes before they enter the main program. Choose one of the following options:

   Stop the processes

   Choose this option if you have not saved a breakpoint file for the current program and you want to set breakpoints before the program runs.

   Let the processes run

   If you have run a program and it has crashed, then running under TotalView and choosing this option will cause the program to crash again, except that, this time, TotalView will show you where the program failed.

4.4.2 Attaching to an Executing Parallel Job

Use the following procedure to attach to a parallel job that is already executing:

1. Start TotalView without using any arguments, as follows:

   ```
   user@tazmo: totalview
   ```

2. In the root window, select the following option:

   ```
   Show All Unattached Processes (N)
   ```

   The unattached processes window is displayed, showing a list of processes to which you can attach.

3. Select the `prun` command in this window to attach to the parallel job as a whole.

4.4.3 Restarting a Parallel Job

You can kill a job and restart from the beginning as follows. Restarting a program is faster than the initial program startup. This is because the TotalView servers remain in place and do not have to be restarted.

1. Select the following option:

   ```
   Arguments/Create/Signal -> Restart Program
   ```

   The initial `prun` process and all the parallel processes are terminated. The `prun` process is restarted to spawn the parallel program again.

2. If you want to preserve breakpoints in your code, select the following option to save them to a file before you restart the program:

   ```
   STOP/BARR/EVAL/GIST -> Save All Action Points
   ```

   TotalView automatically reloads the breakpoints when it restarts the program.

4.4.4 Problems and Limitations

TotalView starts up remote servers on each node on which your parallel program runs. By default, it uses `rsh(1)` when it starts these servers. You must ensure that you can `rsh` to the nodes on which your parallel program runs for this to work.

You can change the command used to start the remote servers using the `Server Launch Window` command in the TotalView root window. See the section *Changing the Option* in Chapter 4 of the *TotalView User's Guide*.

4.5 Using Vampir

You can use Vampir version 2.0 and Vampirtrace version 1.5 to prepare, build, trace and analyze MPI programs running on Compaq AlphaServer SC systems. Vampir is a visualization and analysis tool for MPI programs. Vampirtrace is the Vampir MPI profiling library. The Vampir software is available from Pallas GmbH. Their web site is `http://www.pallas.com`

After installing Vampir, you link your MPI program with Vampirtrace. When you run the program, Vampirtrace uses the MPI profiling interface to gather information about the program's execution behavior. The information is kept locally in each processor's memory and saved in a trace file when the program exits. The trace file is then fed into Vampir for analysis.

4.5.1 Preparing to Use Vampir

Vampir requires that you setup the following environment variables.

1. Set the environment variable `PAL_ROOT` to the directory where you have installed the kits and licenses, for example:

   ```
   user@tazmo: setenv PAL_ROOT /usr/local/vampir
   ```

2. Set the environment variable `DISPLAY` to the node on which you want Vampir to display its graphics. Make sure that you have permission to display on that node.

4.5.2 Linking and Tracing a Program

Use the following steps to link and trace your program:

1. Link your program with the Vampirtrace library and the MPI profiling interface. For C programs, the command line is as follows:

   ```
   cc -o myprog myprog.o -L/usr/local/vampir/lib/lib -1VT -lpmpi -lmpi -lelan
   ```

   For Fortran programs, the command line is as follows:

   ```
   f77 -o test -test.o -L/usr/local/vampir/lib/lib -lfmpi -lVT -lpmpi -lmpi -lelan
   ```

2. Run your program to generate a trace file for Vampir to use:

   ```
   user@tazmo: prun -N 2 myprog
   ...
   Writing logfile myprog.bpv
   Finished writing file
   ```

   The trace file has the same name as the program with a `.bpv` suffix, for example `myprog.bpv`.

3. Use Vampir to view and analyze the trace, as follows:

   ```
   user@tazmo: vampir myprog.bpv
   ```

4.6 MPI Example

The `mping` program uses the MPI library to synchronize the processes and to perform interprocess communications.

4.6.1 MPI Functions

The following functions from this library are used and the header file `mpi.h` is included to declare them.

1. `MPI_Init()` initializes the process to use the library.
2. `MPI_Comm_rank()` establishes the rank or number of the process within the set of parallel processes.
3. `MPI_Comm_size()` determines the number of processes in the parallel program.
4. `MPI_Barrier()` synchronizes all the processes.
5. `MPI_WTime()` reads the value of a timer, which counts in seconds.
6. `MPI_Recv()` receives a message.
7. `MPI_Send()` sends a message.

We will look at these functions more closely when we see them in context.

4.6.2 Command Line Interface

This is the command line interface for the program, `mping`.

```
mping -n number[k|K|m|M] -eh nob [maxNob [incNob]]
```

The options for the programs are:

`-n` *`number`*`[k|K|m|M]`

Specifies the number of times to ping. The *`number`* may have a `k` or an `m` appended to it (or their uppercase equivalents) to denote multiples of 1024 and 1,048,576 respectively. By default, the program pings 100,000 times.

`-e` Instructs every process to print its timing statistics.

`-h` Displays the list of options.

`nob` `[`*`maxNob`* `[`*`incNob`*`]]`

`nob` specifies to `mping` how many bytes there are in each packet. If *`maxNob`* is given, it specifies a maximum number of bytes to send in each packet and invokes the following behavior. After each *`n`* repetitions (as specified with the `-n` option), the packet size is

increased by `incNob` (the default is a doubling in size) and another set of repetitions is performed until the packet size exceeds `maxNob`. This means that if neither of the optional parameters are specified, only one set of repetitions is performed.

4.6.3 Program Output

At the start of the program, if printing has been enabled for all processes by specifying the `-e` option, a message like this is displayed by each process.

```
1(8): MPI PING reps 250000 minNob 64 maxNob 128 incNob 32
```

where `1` is the identity number of the process and `8` gives the number of processes running in parallel.

After each set of repetitions, timing statistics are displayed like this:

```
1 pinged 0: 64 bytes 000000021.25 uSec 00000000.50 MB/s
```

This indicates that process `1` pinged process `0` with 64 byte packets. The pinging took 21.25 μseconds at a rate of 0.5MBytes per second.

If printing has been enabled for all processes with the `-e` option, this message is displayed by each process. By default, only one process in each pair displays the message.

4.6.4 Header Files and Variables

The header files and variables used by the program are shown here. The variables are declared in `main()`.

```
#include <stdio.h> 1
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/time.h>

#include <mpi.h>

int main(int argc, char *argv[]) {

    double      t, tv[2]; 2

    MPI_Status  status; 3
    int         tag = 0x69;
    char        *rbuf, *tbuf;
```

```
    int         reps = 100000; [4]
    int         minNob = 1;
    int         maxNob = 1;
    int         incNob, nob;

    int         proc, peer, nproc; [5]

    int         doprint = 0; [6]
    char        *progname;
    int         c, r, i;
    ...
}
```

The header files and variables are described here.

[1] Besides the standard C header files, the `mpi.h` header file is required for the MPI library.

[2] The two time variables are used to time each set of repetitions of sending and receiving a message. The `tv` array is used to record two time readings, returned by the MPI function `MPI_Wtime()`:

1. The time before the set of repetitions begins.
2. The time after the set of repetitions has ended.

The variable `t` is used to hold the difference between these two readings, converted to microseconds.

[3] The `status` variable is used to determine the status of a message received by a call to `MPI_Recv()`.

The `tag` is used to identify the messages transmitted through the MPI message passing functions. `tbuf` and `rbuf` are pointers to buffers for messages, for which the process allocates space before calling `MPI_Send()` and `MPI_Recv()` to send and receive messages.

[4] This group of variables is used to control how many times the process pings its opposite number and the size of packets sent. The variable `reps` is set to the number of repetitions requested with the `-n` option. It has a default setting of 100,000.

The next three variables hold the minimum, maximum and increment values for the packet size. They are used when more than one set of repetitions is requested. The variable `nob` is used to iterate from `minNob` to `maxNob` during a set of repetitions.

5 These variables are used to identify, by means of their rank, the process and its peer (or opposite number), to which it sends packets, and to hold the total number of processes.

6 The variable `doprint` is used to enable (`1`) or disable (`0`) the printing of results by all the processes.

The `progName` variable is used to extract the name of the program for use with the standard UNIX style `-h` option and `Usage` message, which is displayed when the program is called with the wrong arguments.

The remaining three variables are general purpose iteration variables.

4.6.5 Argument Checking

The first section of `main()` is concerned with checking the arguments passed to the program on the command line.

```
int main(int argc, char *argv[]) {
    ...
    for (progName = argv[0] + strlen (argv[0]); 1
        progName > argv[0] && *(progName - 1) != '/';
        progName--)
        ;

    while ((c = getopt (argc, argv, "n:eh")) != -1) 2
        switch (c) {
        case 'n':
            if ((reps = getSize (optarg)) <= 0)
                usage (progName);
            break;

        case 'e':
            doprint++;
            break;

        case 'h':
            help (progName);

        default:
            usage (progName);
        }

    if (optind == argc) 3
        minNob = 0;
    else if ((minNob = getSize (argv[optind++])) < 0)
        usage (progName);
```

```
    if (optind == argc)
        maxNob = minNob;
    else if ((maxNob = getSize (argv[optind++])) < minNob)
        usage (progName);

    if (optind == argc)
        incNob = 0;
    else if ((incNob = getSize (argv[optind++])) < 0)
        usage (progName);
    ...
}
```

1 The program name is passed in as `argv[0]`, the first string on the command line. This string may take the form of a pathname, such as `/opt/rms/examples/mping`. The `progname` variable is set to point to the end of the program name. The loop then steps the variable backwards, one character at a time, until either a filename separator (`/`) or the beginning of the name is reached. This leaves `progname` pointing at the start of the program name.

2 The `while` loop steps through the options given on the command line.

- If the `-n` option has been used, the variable `reps` is set to the requested number of repetitions after a check that the number is greater than `0`. If the number is invalid, the `usage()` function is called. This merely displays the command line syntax for the program and then exits.
- If the `-e` option has been used, the variable `doprint` is incremented. This variable is used later to enable or disable the printing of statistics.
- The `-h` option calls the `help()` function, which displays the command line syntax for the program and explains the meaning of the various options (or flags), like this.

```
Usage: mping [flags] nob [maxNob] [incNob]
   Flags may be any of:
      -n number         repetitions to time
      -e                everyone print timing info
      -h                print this info

   Numbers may be postfixed with 'k' or 'm'
```

- If any other options besides the three mentioned here are given, the function `usage()` is called to display the correct command line syntax and then exit.

3 The three `if` statements determine whether the optional arguments for specifying a varying packet size have been set. The variable `optind` is defined externally and included by the header files at the start of the program. After stepping through all the options with the `while` loop, `optind` indexes the first argument in `argv`.

The first argument should be `nob`, the number of bytes in each packet. If the user has not specified this argument, the program continues rather than exiting but assumes a value of `0`. Note that the value is assigned to `minNob` rather than to the variable `nob`. Later on, the value is transferred to `nob` when it acts as an iteration variable.

4.6.6 Initialization

The next section of `main()` is concerned with initializing the process to use the MPI library.

```
int main(int argc, char *argv[]) {
    ...
    MPI_Init(&argc, &argv); 1
    MPI_Comm_rank(MPI_COMM_WORLD, &proc); 2
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);

    if (nproc == 1)
       exit(1);

    if ((rbuf = (char *)malloc(maxNob ? maxNob : 8)) == NULL) { 3
        perror ("Failed memory allocation");
        exit (1);
    }

    if ((tbuf = (char *)malloc(maxNob ? maxNob : 8)) == NULL) {
        perror ("Failed memory allocation");
        exit (1);
    }

    for (i = 0; i < maxNob; i++)
        tbuf[i] = i & 255;
    ...
}
```

The initialization process is as follows.

1 The process calls `MPI_Init()` to initialize itself to use the MPI library. The function allocates and initializes a structure referenced by the opaque handle `MPI_COMM_WORLD`. This handle is required as a parameter to the other functions used.

2 The process calls `MPI_Comm_rank()` to determine its rank. The processes are numbered from `0` to `nproc-1` where `nproc` is the number of processes running in parallel, as established by a call to `MPI_Comm_size()`. If there is only one process, the process exits as there is no other process that it can ping.

3 The process allocates memory for the two message buffers using `malloc`. The buffers are used as the source and destination of the messages that are transferred across the network. Pointers to them are passed to the MPI message passing library functions.

If a maximum number of bytes for the packet size is specified on the command line to `mping`, the process allocates a buffer of this size. By default, the buffers hold 8 bytes. The transmit buffer is initialized by writing a sequence of numbers to it.

4.6.7 Establishing the Peer Group

Before starting the first (and possibly only) set of repetitions, the processes must synchronize and group themselves into pairs.

```
int main(int argc, char *argv[]) {
    ...
    if (doprint) 1
        printf ("%d(%d): MPI PING reps %d"
                "minNob %d maxNob %d incNob %d\n",
                proc, nproc, reps, minNob, maxNob, incNob);

    MPI_Barrier(MPI_COMM_WORLD); 2

    peer = proc ^ 1; 3
    if (peer >= nproc)
        doprint = 0;
    ...
}
```

1 If all the processes have been enabled for printing with the `-e` option, each prints a message to confirm its identity, the number of processes in the program and the program parameters.

2 Before starting to ping each other, the processes synchronize, that is, each waits in the call to `MPI_Barrier()` until all have made the call. This guarantees that all the processes are initialized and ready to send and receive messages before any one of them starts to ping another.

3 To ping each other, the processes split up into pairs. Each process determines its opposite number or `peer` simply by an exclusive-OR of its own rank with the constant `1`. With an uneven number of processes, one will have no peer. This can be determined by checking that the peer's rank is in the valid range. This singleton is disabled from printing.

4.6.8 Sending Messages

In the final section of `main`, the process pings its peer a given number of times using the MPI message passing functions.

```
int main(int argc, char *argv[]) {
   ...
   for (nob = minNob;
        nob <= maxNob;
        nob = incNob ? nob + incNob : nob ? 2 * nob : 1) { 1

        r = reps;

        MPI_Barrier(MPI_COMM_WORLD); 2

        tv[0] = MPI_Wtime();

        if (peer < nproc) {   3

            if (proc & 1) {

                r--;
                MPI_Recv(rbuf, nob, MPI_BYTE, peer, tag,
                         MPI_COMM_WORLD, &status);
            }

            while (r-- > 0) { 4

                MPI_Send(tbuf, nob, MPI_BYTE, peer, tag,
                         MPI_COMM_WORLD);

                MPI_Recv(rbuf, nob, MPI_BYTE, peer, tag,
                         MPI_COMM_WORLD, &status);
             }

             if (proc & 1) {

                 MPI_Send(tbuf, nob, MPI_BYTE, peer, tag,
                          MPI_COMM_WORLD);
             }
         }
```

```
        tv[1] = MPI_Wtime(); 5
        t = dt (&tv[1], &tv[0]) * 1000000.0/ (2 * reps);

        MPI_Barrier(MPI_COMM_WORLD);
        printStats (proc, peer, doprint, nob, t);
    }

    MPI_Barrier(MPI_COMM_WORLD); 6
    return (0);
}
```

The MPI library message passing and interval timing functions are described here.

1 The `for` loop controls how many sets of repetitions are performed. In each set of repetitions, a message containing `nob` bytes is sent from one process to its peer for the number of times specified by `reps`.

The first time through the loop, `nob` is set to `minNob`. This was initialized earlier (see Section 4.6.5) to the value the user entered for `nob` on the command line (by default, `0`).

On subsequent iterations, the value of `nob` is incremented by the value of `incNob`. If no value was specified for `incNob` on the command line, the original value of `nob` is doubled or, if `nob` was unspecified, it is set to `1`.

If the user specified `maxNob`, the `for` loop is iterated until `nob` exceeds the value of `maxNob`. If not, the loop is only executed once.

2 Before the processes begin to time how long the ping operation takes, they synchronize using `MPI_Barrier()`. This ensures that they are all ready to start sending and receiving messages at the same time.

The timing is done by taking two readings using the function `MPI_Wtime()`, which returns the value of a timer in seconds: one reading before the messages start and one when they have finished.

3 After testing that the process has a peer (this test has to be repeated in here since all the processes must participate in the synchronization), the message sending can begin. The odd-numbered processes (`proc & 1`) start first by issuing a receive command.

The call to `MPI_Recv()` causes the process to block waiting for the arrival of a message from the sender with a rank of `peer` and with a message tag of `tag`. `MPI_COMM_WORLD` specifies the communications group to which the process and its `peer` belong.

The `rbuf` parameter to `MPI_Recv()` points to a destination buffer for the message while `nob` specifies its size in units with a datatype of `MPI_BYTE`.

The `status` parameter is filled in on return to indicate details of the transfer. In this application, the value of `status` is not checked.

4 In the `while` loop both the odd and even-numbered processes send a message and then wait for a reply, decrementing the number of repetitions, `r`, each time. The call to `MPI_Send()` specifies:

- A source buffer and its size in units of datatype `MPI_BYTE`.
- The rank of the destination process (`peer`). Note that the MPI library takes care of all routing details; the sender does not have to know where in the network the receiver resides.
- A message tag to identify the message.
- The `MPI_COMM_WORLD` parameter specifies the group to which the process and its peer belong.

The process blocks in the call to `MPI_Send()` until the message has been received.

Finally, the odd-numbered processes send the last message in the sequence of repetitions. By making the odd-numbered processes request a receive to begin with while the even-numbered processes send a message, deadlock is avoided.

5 After the set of repetitions, the process reads the timer again. It calculates the time taken for one ping in each direction (the difference between the two timer readings divided by the number of repetitions). This value is converted to microseconds (multiplied by 1,000,000) and halved to get the value for a ping in one direction.

Before the processes print the results, they synchronize again. This means that all the results are displayed at roughly the same time and the printing does not interfere with the network performance.

6 When the process has come out of the `for` loop, it synchronizes with its peers again before exiting.

4.6.9 Subsidiary Functions

The subsidiary functions make no use of the MPI library.

`getSize()`	This function checks whether the user has suffixed the number of repetitions, specified on the command line with the `-n` option, with either a `k` or `K` (for kilobytes) or `m` or `M` (for megabytes). If it finds a suffix, it multiplies the number as appropriate (a left shift by one place multiplies by 2).
`dt`	This function returns the difference between its two arguments.
`usage`	This function prints out the command line syntax for the program and then exits.
`help`	This function prints out the command line syntax for the program and enumerates the various options before exiting.
`printStats`	This function displays the timing statistics generated during each set of repetitions. Unless printing is enabled for all processes with the `-e` option, only the odd-numbered processes have their statistics displayed.

4.6.10 Compiling and Running the Program

To compile the program `mping.c`, which uses the MPI library,

```
user@tazmo: cc -o mping mping.c -lmpi -lelan
```

Before we run the program with `prun`, we can find out how many processors are available with `rinfo`, as described in Section 3.4.

```
tony@tazmo1: rinfo
MACHINE        CONFIGURATION
tazmo          day

PARTITION      CPUS    STATUS        TIME   TIMELIMIT  NODES
root              8                                   tazmo[0-3]
parallel        2/8   running    05:00:12             tazmo[0-3]

RESOURCE       CPUS    STATUS        TIME    USERNAME  NODES
parallel.48       2 allocated       00:15      duncan  tazmo0

JOB            CPUS    STATUS        TIME    USERNAME  NODES
parallel.259      2   running       00:04      duncan  tazmo0
```

Here we see that the partition called `parallel` is active. There are eight processors in this partition but two of them are allocated to the user called `duncan` who is running a job identified by the name `parallel.259`. This leaves six processors free.

Using the command `prun`, we can get four of these processors allocated to us and run `mping` on each of them.

```
user@tazmo: prun -p parallel -n 4 mping -e
```

By giving the `-e` option to `mping`, we can see what the differences in timing are between the two pairs of processes. If four nodes were available, we could request that the program be run one process per node with the `-N` option to `prun`.

```
user:tazmo: prun -p parallel -N 4 mping -e
```

At this point you might like to experiment with running the program on different combinations of nodes. You will see differences in latency and bandwidth depending upon whether the communicating processes are on the same node or different nodes.

4.7 Shmem Example

The `sping` program uses the Shmem library routines to synchronize the processes and to perform interprocess communications.

4.7.1 Shmem Functions

The following functions from this library are used and the header file `shmem.h` is included to declare them.

1. `shmem_init()` initializes the process to use the library.
2. `shmem_barrier_all()` synchronizes all the processes.
3. `shmem_put()` sends a message.
4. `shmem_wait()` waits until the value of a flag, supplied as an argument, changes.
5. `my_pe()` returns the rank or number of the process within the set of parallel processes.
6. `num_pes()` returns the number of processes in the program.

The C source code for this example is in the file `sping.c` in the directory `/opt/rms/examples`.

4.7.2 Command Line Interface

This is the command line interface for the program, `sping`.

```
sping -n number[k|K|m|M] -eh nwords [maxWords [incWords]]
```

The options for the programs are:

`-n` *number*`[k|K|m|M]`

Specifies the number of times to ping. The *number* may have a `k` or an `m` appended to it (or their uppercase equivalents) to denote multiples of 1024 and 1,048,576 respectively. By default, the program pings 10,000 times.

`-e` Instructs every process to print its timing statistics.

`-h` Displays the list of options.

nwords `[`*maxWords* `[`*incWords*`]]`

nwords specifies to `sping` how many words there are in each packet. If *maxWords* is given, it specifies a maximum number of words to send in each packet and invokes the following behavior. After each *n* repetitions (as specified with the `-n` option), the packet size is increased by `incWords` (the default is a doubling in size) and another set of repetitions is performed until the packet size exceeds `maxWords`. This means that if neither of the optional parameters are specified, only one set of repetitions is performed.

4.7.3 Program Output

At the start of the program, if printing has been enabled for all processes by specifying the `-e` option, a message like this is displayed by each process.

```
1(8): Shmem PING reps 10000 minWords 1 maxWords 256 incWords 0
```

where `1` is the identity number of the process and `8` gives the number of processes running in parallel.

After each set of repetitions, timing statistics are displayed like this:

```
  1 pinged   0:        1 words      3.12 uSec     2.56 MB/s
  1 pinged   0:        2 words      3.12 uSec     5.12 MB/s
  1 pinged   0:        4 words      3.22 uSec     9.93 MB/s
  1 pinged   0:        8 words      3.42 uSec    18.72 MB/s
  1 pinged   0:       16 words      3.81 uSec    33.61 MB/s
  1 pinged   0:       32 words      4.39 uSec    58.25 MB/s
  1 pinged   0:       64 words      4.49 uSec   113.98 MB/s
  1 pinged   0:      128 words      7.47 uSec   137.07 MB/s
  1 pinged   0:      256 words     14.41 uSec   142.10 MB/s
```

This indicates that when process `1` pinged process `0` with 64 word packets. The operation took 4.49 μseconds at a rate of 113.98MBytes per second.

If printing has been enabled for all processes with the `-e` option, this message is displayed by each process. By default, only one process in each pair displays the message.

4.7.4 Header Files and Variables

The header files and variables used by the program are shown here. The variables are declared in `main()`.

```
#include <stdio.h> [1]
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/time.h>

#include <elan/shmem.h>

int main(int argc, char *argv[]) {

    double      t, tv[2]; [2]

    int         tag = 0x69; [3]
    long       *rbuf, *tbuf;

    int         reps = 10000; [4]
    int         minWords = 1;
    int         maxWords = 1;
    int         incWords, nwords;

    int         proc, peer, nproc; [5]

    int         doprint = 0; [6]
    char        *progname;
    int         c, r, i;
    ...
}
```

The header files and variables are described here.

[1] Besides the standard C header files, the `shmem.h` header file is required for the Shmem library.

[2] The two time variables are used to time each set of repetitions of sending and receiving a message. The `tv` array is used to record two time readings:

1. The time before the set of repetitions begins.

2. The time after the set of repetitions has ended.

The variable `t` is used to hold the difference between these two readings.

3 The `tag` variable is used to identify the messages sent through the Shmem communications functions. `tbuf` and `rbuf` are pointers to buffers for the messages, for which the process allocates space before calling `shmem_put()` and `shmem_wait()` to send and receive messages.

4 This group of variables is used to control how many times the process pings its opposite number and the size of packets sent. The variable `reps` is set to the number of repetitions requested with the `-n` option. It has a default setting of 10,000.

The next three variables hold the minimum, maximum and increment values for the packet size. They are used when more than one set of repetitions is requested. The variable `nwords` is used to iterate from `minWords` to `maxWords` during a set of repetitions.

5 These variables are used to identify, by means of their rank, the process and its peer (or opposite number), to which it sends packets, and to hold the total number of processes.

6 The variable `doprint` is used to enable (`1`) or disable (`0`) the printing of results by all the processes.

The `progName` variable is used to extract the name of the program for use with the standard UNIX style `-h` option and `Usage` message, which is displayed when the program is called with the wrong arguments.

The remaining three variables are general purpose iteration variables.

4.7.5 Argument Checking

The first section of `main()` is concerned with checking the arguments passed to the program on the command line.

```
int main(int argc, char *argv[]) {
    ...
    for (progName = argv[0] + strlen (argv[0]); 1
         progName > argv[0] && *(progName - 1) != '/';
         progName--)
         ;
```

```
    while ((c = getopt (argc, argv, "n:eh")) != -1) 2
        switch (c) {
        case 'n':
            if ((reps = getSize (optarg)) <= 0)
                usage (progName);
            break;

        case 'e':
            doprint++;
            break;

        case 'h':
            help (progName);

        default:
            usage (progName);
        }

    if (optind == argc) 3
        minWords = 1;
    else if ((minWords = getSize (argv[optind++])) < 0)
        usage (progName);

    if (optind == argc)
        maxWords = minWords;
    else if ((maxWords = getSize (argv[optind++])) < minWords)
        usage (progName);

    if (optind == argc)
        incWords = 0;
    else if ((incWords = getSize (argv[optind++])) < 0)
        usage (progName);
    ...
}
```

1 The program name is passed in as `argv[0]`, the first string on the command line. This string may take the form of a pathname, such as `/opt/rms/examples/sping`. The `progname` variable is set to point to the end of the program name. The loop then steps the variable backwards, one character at a time, until either a filename separator (`/`) or the beginning of the name is reached. This leaves `progname` pointing at the start of the program name.

2 The `while` loop steps through the options given on the command line.

- If the `-n` option has been used, the variable `reps` is set to the requested number of repetitions after a check that the number is greater than `0`. If the number is invalid, the `usage()` function is

called. This merely displays the command line syntax for the program and then exits.

- If the `-e` option has been used, the variable `doprint` is incremented. This variable is used later to enable or disable the printing of statistics.
- The `-h` option calls the `help()` function, which displays the command line syntax for the program and explains the meaning of the various options (or flags), like this.

```
Usage: sping [flags] nwords [maxWords] [incWords]
   Flags may be any of:
      -n number         repetitions to time
      -e                everyone print timing info
      -h                print this info

   Numbers may be postfixed with 'k' or 'm'
```

- If any other options besides the three mentioned here are given, the function `usage()` is called to display the correct command line syntax and then exit.

3 The three `if` statements determine whether the optional arguments for specifying a varying packet size have been set. The variable `optind` is defined externally and included by the header files at the start of the program. After stepping through all the options with the `while` loop, `optind` indexes the first argument in `argv`.

The first argument should be `nwords`, the number of words in each packet. If the user has not specified this argument, the program continues rather than exiting but assumes a value of `1`. Note that the value is assigned to `minWords` rather than to the variable `nwords`. Later on, the value is transferred to `nwords` when it acts as an iteration variable.

4.7.6 Initialization

The next section of `main()` is concerned with initializing the process to use the Shmem library.

```
int main(int argc, char *argv[]) {
    ...
    shmem_init(); 1

    proc = my_pe(); 2
    nproc = num_pes();
```

```
    if (nproc == 1)
       exit(1);

    if (!(rbuf = (long *)malloc((maxWords ? maxWords : 1) * sizeof(long)))) { 3
        perror ("Failed memory allocation");
        exit (1);
    }

    if (!(tbuf = (long *)malloc((maxWords ? maxWords : 1) * sizeof(long)))) {
        perror ("Failed memory allocation");
        exit (1);
    }

    for (i = 0; i < maxWords; i++) 4
        tbuf[i] = 1000 + (i & 255);
    *rbuf = 0;
    ...
}
```

The initialization process is as follows.

1 The process calls `shmem_init()` to initialize itself to use the Shmem library.

2 The process uses the function `my_pe()` to determine its rank and `num_pes()` to find out how many processes are running in parallel. Both functions are from the Shmem library. The processes are numbered from `0` to `nproc-1`, where `nproc` is established by the call to `num_pes`. If there is only one process, the process exits as there is no other process that it can ping.

3 The process allocates memory for the two message buffers using `malloc`. The buffers are used as the source and destination of the messages that are transferred across the network. Pointers to them are passed to the Shmem library communications functions.

If a maximum number of words for the packet size is specified on the command line to `sping`, the process allocates a buffer of this size. By default, the buffers are 1 word in size.

4 The transmit buffer is initialized by writing a sequence of numbers to it, starting at `1000`. The first word of the receive buffer is initialized to `0`. The reason for this is explained in Section 4.7.8.

4.7.7 Establishing the Peer Group

Before starting the first (and possibly only) set of repetitions, the processes must synchronize and group themselves into pairs.

```
int main(int argc, char *argv[]) {
    ...
    if (doprint) [1]
        printf ("%d(%d): Shmem PING reps %d"
                "minWords %d maxWords %d incWords %d\n",
                proc, nproc, reps, minWords, maxWords, incWords);

    shmem_barrier_all(); [2]

    peer = proc ^ 1; [3]
    if (peer >= nproc)
        doprint = 0;
    ...
}
```

[1] If all the processes have been enabled for printing with the `-e` option, each prints a message to confirm its identity, the number of processes in the program and the program parameters.

[2] Before starting to ping each other, the processes synchronize, that is, each waits in the call to `shmem_barrier_all()` until all have made the call. This guarantees that all the processes are initialized and ready to send and receive messages before any one of them starts to ping another.

[3] To ping each other, the processes split up into pairs. Each process determines its opposite number or `peer` simply by an exclusive-OR of its own rank with the constant `1`. With an uneven number of processes, one will have no peer. This can be determined by checking that the peer's rank is in the valid range. This singleton is disabled from printing.

4.7.8 Sending Messages

In the final section of `main`, the process pings its peer a given number of times using the Shmem communications functions.

```
int main(int argc, char *argv[]) {
   ...
   for (nwords = minWords;
        nwords <= maxWords;
        nwords = incWords ? nwords + incWords :
        nwords ? 2 * nwords : 1) { [1]

        r = reps;

        shmem_barrier_all(); [2]
```

```
        tv[0] = gettime();

        if (peer < nproc) {   3

            if (proc & 1) {
                r--;
                shmem_wait(rbuf, 0);
                *rbuf = 0;
            }

            while (r-- > 0) { 4

                shmem_put(rbuf, tbuf, nwords, peer);
                shmem_wait(rbuf, 0);
                *rbuf = 0;
             }

             if (proc & 1) {
                shmem_put(rbuf, tbuf, nwords, peer);
             }
         }

         tv[1] = gettime(); 5
         t = dt (&tv[1], &tv[0]) / (2 * reps);

         shmem_barrier_all();
         printStats (proc, peer, doprint, nwords, t);
    }
    shmem_barrier_all(); 6
    return (0);
}
```

The Shmem library communications and interval timing functions are described here.

1 The `for` loop controls how many sets of repetitions are performed. In each set of repetitions, a message containing `nwords` words is sent from one process to its peer for the number of times specified by `reps`.

The first time through the loop, `nwords` is set to `minWords`. This was initialized earlier (see Section 4.7.5) to the value the user entered for `nwords` on the command line (by default, `1`).

On subsequent iterations, the value of `nwords` is incremented by the value of `incWords`. If no value was specified for `incWords` on the command line, the original value of `nwords` is doubled.

If the user specified `maxWords`, the `for` loop is iterated until `nwords` exceeds the value of `maxWords`. If not, the loop is only executed once.

2 Before the processes begin to time how long the ping operation takes, they synchronize using `shmem_barrier_all()`. This ensures that they are all ready to start sending and receiving messages at the same time.

The timing is done by taking two readings using the function `gettime()`, which returns the current time in microseconds: one before the messages start and one when they have finished.

3 After testing that the process has a peer (this test has to be repeated in here since all the processes must participate in the synchronization), the message sending can begin. The odd-numbered processes (`proc & 1`) start first by issuing a receive command.

The call to `shmem_wait()` waits until the value in the first word of the buffer pointed to by `rbuf` differs from the second argument, which is set to `0`. During initialization (see Section 4.7.6), the first word of the the buffer was set to `0` while the first word of the transmit buffer `tbuf` was set to `1000`.

As soon as `shmem_wait()` returns, the value of the first word in the receive buffer is reset to `0`, ready for the next time.

4 In the `while` loop both the odd and even-numbered processes send a message and then wait for a reply, decrementing the number of repetitions, `r`, each time. The call to `shmem_put()` specifies:

- A destination buffer.
- A source buffer.
- The length of the message in words.
- The rank of the destination process. Note that the Shmem library takes care of all routing details; the sender does not have to know where in the network the receiver resides.

The function blocks until the message has been transferred to the receiver's buffer. Then, the process calls `shmem_wait()` to receive its message.

Finally, the odd-numbered processes send the last message in the sequence of repetitions. By making the odd-numbered processes request a receive to begin with while the even-numbered processes send a message, deadlock is avoided.

5 After the set of repetitions, the process reads the timer again. It calculates the time taken for one ping in each direction (the difference

between the two timer readings divided by the number of repetitions). This value (expressed in microseconds) is halved to get the value for a ping in one direction.

Before the processes print the results, they synchronize again. This means that all the results are displayed at roughly the same time and the printing does not interfere with the network performance.

6 When the process has come out of the `for` loop, it synchronizes with its peers again before exiting.

4.7.9 Subsidiary Functions

The subsidiary functions make no use of the Shmem library.

`getSize()` This function checks whether the user has suffixed the number of repetitions, specified on the command line with the `-n` option, with either a `k` or `K` (for kilobytes) or `m` or `M` (for megabytes). If it finds a suffix, it multiplies the number as appropriate (a left shift by one place multiplies by 2).

`gettime` This function returns the current time in microseconds.

`dt` This function returns the difference between its two arguments.

`usage` This function prints out the command line syntax for the program and then exits.

`help` This function prints out the command line syntax for the program and enumerates the various options before exiting.

`printStats` This functions displays the timing statistics generated during each set of repetitions. Unless printing is enabled for all processes with the `-e` option, only the odd-numbered processes have their statistics displayed.

4.7.10 Compiling and Running the Program

To compile the program `sping.c`, which uses the Shmem library, we use the C compiler as shown here.

```
cc -o sping sping.c -lshmem -lelan
```

The file `shmem.c` contains simple implementations of the Shmem library routines. These are compiled with `sping.c`. `shmem.c` uses functions from the Elan library, which is referenced with the `-l` flag.

Before we run the program with `prun`, we can find out how many processors are available with `rinfo`, as described in Section 3.4.

```
tony@tazmo1: rinfo
MACHINE        CONFIGURATION
tazmo          day

PARTITION      CPUS    STATUS        TIME   TIMELIMIT  NODES
root              8                                   tazmo[0-3]
parallel        2/8   running    05:00:12             tazmo[0-3]

RESOURCE       CPUS    STATUS        TIME    USERNAME  NODES
parallel.48       2 allocated       00:15      duncan  tazmo0

JOB            CPUS    STATUS        TIME    USERNAME  NODES
parallel.259      2   running       00:04      duncan  tazmo0
```

Here we see that the partition called `parallel` is active. There are eight processors in this partition but two of them are allocated to the user called `duncan` who is running a job identified by the name `parallel.259`. This leaves six processors free. Using the command `prun`, we can get four of these processors allocated to us and run `sping` on each of them.

```
prun -p parallel -n 4 sping -e
```

By giving the `-e` option to `sping`, we can see what the differences in timing are between the two pairs of processes. If four nodes were available, we could run the program one process per node by using the `-N` option to `prun`

```
prun -p parallel -N 4 sping -e
```

A

RMS Commands

A.1 Overview

The RMS user commands are described in alphabetical order in this appendix. They are as follows:

`allocate`	The `allocate` program reserves access to a set of resources either for running multiple tasks in parallel or for running a sequence of commands on the same CPUs.
`prun`	The `prun` program loads and runs parallel programs. It can also run multiple copies of a sequential program.
`rinfo`	The `rinfo` program displays information about the resources available and about the jobs which are running.
`rmsexec`	The `rmsexec` program runs a sequential program on a lightly loaded node.
`rmsquery`	The `rmsquery` program submits SQL queries to the database. The queries can extract information from the database but cannot update it.

NAME

`allocate` – reserves access to CPUs

SYNOPSIS

allocate [-hiv] [-B *basenode*] [-C *CPUs*] [-N *nodes*] [-p *partition*] [*script* [*args ...*]]

OPTIONS

`-B` *basenode* Specifies the number of the base node (the first node to use) in the partition. Numbering within the partition starts at `0`. By default the base node is unassigned, leaving the scheduler free to select nodes that are not in use.

`-C` *CPUs* Specifies the number of CPUs required per node (default 1).

`-h` Display the list of options.

`-i` Allocate CPUs immediately or fail. By default, `allocate` blocks until resources become available.

`-N` *nodes* | *all*

Specifies the number of nodes to allocate (default 1). You may allocate all nodes in the partition using the argument `all` (i.e `allocate -N all`).

`-p` *partition* Specifies the target partition from which the resources are to be allocated.

`-v` Specifies verbose operation.

DESCRIPTION

The `allocate` program allocates resources for subsequent use by the `prun(1)` command. The `-N`, `-C` and `-B` options control which resources are allocated. A contiguous range of nodes is allocated to the request.

The Partition Manager, `pmanager` allocates processing resources to users as and when the resources are requested and become available . The `allocate` command should be

used when a user wants to run a sequence of commands or several programs concurrently on the same set of CPUs.

The *script* argument (with optional arguments) can be used in two different ways, as follows:

1. *script* is not specified, in which case an interactive command shell is spawned with the resources allocated to it. The user can confirm that resources have been allocated to an interactive shell by using the `rinfo` command (see Page A-11).

   The resources are reserved until the shell exits or until a time limit defined by the system administrator expires, whichever happens first.

   Parallel programs, executed from this interactive shell, all run on the shell's resources (concurrently, if sufficient resources are available).

2. *script* specifies a shell script, in which case the resources are allocated to the named sub-shell and freed when execution of the script completes.

ENVIRONMENT VARIABLES

The following environment variables may be used to identify resource requirements and modes of operation to `allocate`. These environment variables will be used where no specific command line options are specified.

`RMS_IMMEDIATE` Controls whether to exit rather than block if resources are not immediately available. By default, `allocate` blocks until resources become available. Root resource requests are always met.

`RMS_MEMLIMIT` The maximum amount of memory required. This must be less than or equal to the limit set by the system administrator. .

`RMS_PARTITION` Specifies the name of a partition. This is the same as using the `-p` option.

`RMS_PROJECT` The name of the project with which the request should be associated for accounting purposes.

`RMS_TIMELIMIT` Specifies the execution time limit in seconds. The program will be signalled either after this time has elapsed or after any time limit imposed by the system has elapsed. The shorter of the two time limits is used.

`RMS_DEBUG` Whether to execute in verbose mode and display diagnostic messages. This is the same as using the `-v` option. Setting a value of `1` or more will generate additional information that may be useful in diagnosing problems.

Information on the current priority level, project name and memory limit is accessible through the command `rinfo` (see Page A-11) using the `-l` option.

`allocate` passes all existing environment variables through to the shell that it executes. In addition, it sets the following environment variable:

`RMS_RESOURCEID` The ID of the allocated resource.

EXAMPLES

To run a sequence of jobs on the same CPUs

```
duncan@gold0: allocate -N16 jobscript
```

Where `jobscript` is a shell script such as this

```
#!/bin/sh
# simple job script
prun -n16 program1
prun -n16 program2
```

If the script was run directly then each resource request would block and there would be no guarantee of using the same CPUs. By running it under allocate there is only one resource request and both jobs are run on the same CPUs.

To run two programs on the same CPUs at the same time

```
duncan@gold0: allocate -N16 -C2 << EOF
prun program1 &
prun program2 &
rinfo
wait
EOF
```

SEE ALSO

`prun`, `rinfo`

NAME

`prun` – runs a parallel program

SYNOPSIS

```
prun [-hiOrstv] [-B basenode] [-c cpus] [-m block|cyclic]
     [-n processes] [-N nodes] [-p partition] program [args ...]
```

OPTIONS

`-B` *`basenode`*
: Specifies the number of the base node (the first node to use) in the partition. Numbering within the partition starts at `0`. By default the base node is unassigned, leaving the scheduler free to select nodes that are not in use.

`-c` *`cpus`*
: Specifies the number of CPUs required per process (default 1).

`-h`
: Display the list of options.

`-i`
: Allocate CPUs immediately or fail. By default, `prun` blocks until resources become available.

`-n` *`processes`*
: Specifies the number of processes required. The `-n` and `-N` options can be combined to control how processes are distributed over nodes. If neither is specified `prun` starts two processes.

`-N` *`nodes | all`*
: Specifies the number of nodes required. You may also allocate all nodes in a partition using the `all` argument (i.e. `prun -N all`). If the number of nodes is not specified then the RMS scheduler will allocate one CPU per process on nodes with free CPUs.

`-m` *`block | cyclic`*
: Specifies whether to use block (the default) or cyclic distribution of processes over nodes.

`-O`
: Allows resources to be overcommitted. Set this flag if you want to run more than one process per CPU.

`-p` *`partition`*	Specifies the partition on which the program will be executed. By default, the partition specified in the `attributes` table is used.
`-r`	Run processes using `rsh`. Used for admin operations such as starting and stopping RMS
`-s`	Print stats as job exits.
`-t`	Prefix output with the process number.
`-v`	Specifies verbose operation. Multiple `-v` options increase the level of output, `-vv` shows each stage in running a program and `-vvv` enables debug output from the `rmsloader` processes on each node.

DESCRIPTION

The `prun` program executes multiple copies of the specified *`program`* on a partition. `prun` automatically requests resources for the program unless it is executed from a shell that already has resources allocated to it (see Page A-2) .

The way in which processes are allocated to CPUs is controlled by the `-c`, `-n` and `-N` options. The `-n` option specifies the total number of processes to run. The `-c` option specifies the number of CPUs required per process, this defaults to 1. The `-N` option specifies how many nodes are to be used. If it is not used then the scheduler will select CPUs for the program. If the `-N` option is used the scheduler will allocate a contiguous range of nodes and the same number of CPUs on each node.

The `-p` option specifies the partition to use. If no partition is specified then the default partition is used. The default partition is stored in the `attributes` table. Note that use of the `root` partition (all nodes in the machine) is restricted to admin users.

The `-B` option specifies the id of the first node to run the job on. It should be used if you require access to a specific filesystem or device that is not available on all nodes. If the `-B` option is used the scheduler will allocate a contiguous range of nodes and the same number of CPUs on each node. Using this option will cause a request to block until this node and any additional nodes required to run the program are free.

The `-i` option specifies that resource requests should fail if they cannot be met immediately.

The `-m` option specifies how processes are to be distributed over nodes. The choice is between `block` (the default) and `cyclic`. If a program has `n` processes with ids `0,1,...n-1`distributed over `N` nodes then in a block distribution the first `n/N` processes will be allocated to the first node and so on. If the distribution is cyclic, process 0 runs on the first node, process 1 on the second and so on until we run out of nodes. at which stage the distribution wraps – with process `N` running on the first node and so on.

prun exits when all of the processes in the parallel program have exited or when one process has been killed. If all processes exit cleanly then the exit status of prun is the global OR of their individual exit status values. If one of the processes is killed prun will exit with a status value of 128 plus the signal number. prun can also exit with the following codes:

125 One or more processes were still running when the exit timeout expired.

126 prun was run with the -i option and resources were not available.

127 prun was run with invalid arguments.

If an application process started by prun is killed RMS will run a postmortum analysis script that generates a backtrace if it can find a core file for the process.

ENVIRONMENT VARIABLES

The following environment variables may be used to identify resource requirements and modes of operation to prun. These environment variables will be used where no specific command line options are specified.

RMS_IMMEDIATE Controls whether to exit rather than block if resources are not immediately available. By default, prun blocks until resources become available. Root resource requests are always met.

RMS_MEMLIMIT The maximum amount of memory required. This must be less than or equal to the limit set by the system administrator. (see Page A-11). .

RMS_PARTITION Specifies the name of a partition. This is the same as using the -p option.

RMS_PROJECT The name of the project with which the job should be associated for accounting purposes.

RMS_TIMELIMIT Specifies the execution time limit in seconds. The program will be signalled either after this time has elapsed or after any time limit imposed by the system has elapsed. The shorter of the two time limits is used.

RMS_DEBUG Whether to execute in verbose mode and display diagnostic messages. This is the same as using the -v option. Setting a value of 1 or more will generate additional information that may be useful in diagnosing problems.

`RMS_EXITTIMEOUT`

Specify the time allowed in seconds between the first process exit and the last. This option can be useful in parallel programs where one process can exit leaving the others blocked in inter-process communication. It should be used in conjunction with an exit barrier at the end of correct execution of the program.

Information on the current priority level, project name and memory limit is accessible through the command `rinfo` (see Page A-11) using the `-l` option.

`prun` passes all existing environment variables through to the processes that it executes. In addition, it sets the following environment variables:

`RMS_JOBID`	The identifier for the job.
`RMS_NNODES`	The number of nodes used by the application.
`RMS_NODEID`	Logical id of the node within the set allocated to the application.
`RMS_NPROCS`	The total number of processes in the application.
`RMS_RANK`	The rank of the process in the application. The rank ranges from `0` to `n-1`, where `n` is the number of processes in the program.
`RMS_RESOURCEID`	The ID of the allocated resource.

EXAMPLES

In this first example, `prun` is used to run a 4-process program with no specification of where the processes should run.

```
duncan@plaguei: prun -n4 hostname
plague0.quadrics.com
plague0.quadrics.com
plague0.quadrics.com
plague0.quadrics.com
```

The machine `plague` has 4 CPUs per node and so by default the scheduler allocates all 4 CPUs on one node to run the program. Adding the `-N` option allows us to control how the processes are distributed over nodes.

```
duncan@plaguei: prun -n4 -N2 hostname
plague0.quadrics.com
plague0.quadrics.com
plague1.quadrics.com
plague1.quadrics.com
```

```
duncan@plaguei: prun -n4 -N4 hostname
plague1.quadrics.com
plague3.quadrics.com
plague0.quadrics.com
plague2.quadrics.com
```

The `-m` option allows us to control how processes are distributed over nodes. It is used here in conjunction with the `-t` which tags each line of output with the id of the process that wrote it.

```
duncan@plaguei: prun -t -n4 -N2 -mblock hostname
0 plague0.quadrics.com
1 plague0.quadrics.com
2 plague1.quadrics.com
3 plague1.quadrics.com
duncan@plaguei: prun -t -n4 -N2 -mcyclic hostname
0 plague0.quadrics.com
2 plague0.quadrics.com
1 plague1.quadrics.com
3 plague1.quadrics.com
```

The examples so far have used simple UNIX® utilities to illustrate where processes are run. Parallel programs are run in just the same way, the following example measures DMA performance between a pair of processes on different nodes.

```
duncan@plaguei: prun -N2 dping 0 1k
  0:        0 bytes      2.33 uSec     0.00 MB/s
  0:        1 bytes      3.58 uSec     0.28 MB/s
  0:        2 bytes      3.61 uSec     0.55 MB/s
  0:        4 bytes      2.44 uSec     1.64 MB/s
  0:        8 bytes      2.47 uSec     3.24 MB/s
  0:       16 bytes      2.55 uSec     6.27 MB/s
  0:       32 bytes      2.57 uSec    12.45 MB/s
  0:       64 bytes      3.48 uSec    18.41 MB/s
  0:      128 bytes      4.23 uSec    30.25 MB/s
  0:      256 bytes      4.99 uSec    51.32 MB/s
  0:      512 bytes      6.39 uSec    80.08 MB/s
  0:     1024 bytes      9.26 uSec   110.55 MB/s
```

The `-s` option instructs `prun` to print a summary of the resources used by the job when it finishes.

```
duncan@plaguei: prun -s -N2 dping 0 32
  0:        0 bytes      2.35 uSec     0.00 MB/s
  0:        1 bytes      3.60 uSec     0.28 MB/s
  0:        2 bytes      3.53 uSec     0.57 MB/s
  0:        4 bytes      2.44 uSec     1.64 MB/s
  0:        8 bytes      2.47 uSec     3.23 MB/s
  0:       16 bytes      2.54 uSec     6.29 MB/s
```

```
  0:       32 bytes      2.57 uSec    12.46 MB/s
Elapsed time         1.00 secs    Allocated time       1.99 secs
User time            0.93 secs    System time          0.13 secs
Cpus used               2
```

Note that the allocated time (in CPU seconds) is twice the elapsed time (in seconds) as two CPUs were allocated.

See Also

`allocate`, `rinfo`

NAME

`rinfo` – Displays resource usage and availability information for parallel jobs

SYNOPSIS

```
rinfo [-achjlmnpqr] [-L [partition] [statistic]]
      [-s daemon|all  [hostname]] [-t node | name]
```

OPTIONS

`-a` List all resources and jobs (both the user's and those of others).

`-c` List the configuration names.

`-h` Display the list of options.

`-j` List current jobs. This can be combined with the `-a` option to get a list of all jobs (both the user's and those of others).

`-l` Give more detailed information.

`-m` Show the machine name.

`-n` Show the status of each node. Can be combined with `-l`.

`-p` Identify each active partition by name and indicate the number of CPUs in each partition.

`-q` Print information on the user's quotas and projects.

`-r` Show the allocated resources.

`-L` *`partition statistic`*

Print the hostname of a lightly loaded node in the machine or the specified *`partition`*. RMS provides a load balancing service, accessible through `rmsexec`, that enables users to run their processes on lightly loaded nodes, where loading is evaluated according to a given *`statistic`* (see `rmsexec` Page A-14) .

`-s` *daemon*|*all* [*hostname*]

Show the status of the *daemon*. When used with the argument *all* `rinfo` will show the status of all daemons running on the `rmshost` management node. For daemons that run on multiple nodes, such as `rmsd`, the optional *hostname* argument specifies the hostname of the node on which the daemon is running.

`-t` *node* | *name*

Where *node* is the network ID of a node, `rinfo` translates it into the hostname; where *name* is a hostname, `rinfo` translates it into the network ID.

DESCRIPTION

The `rinfo` program displays information about resource usage and availability. Its default output is in four parts that identify: the machine, the active configuration, resource requests and the current jobs. Note that the latter sections are only displayed if jobs are active.

```
robin@tazmo1: rinfo
MACHINE        CONFIGURATION
tazmo          day

PARTITION      CPUS    STATUS        TIME   TIMELIMIT  NODES
root              6                                   tazmo[0-2]
parallel        2/4   running    01:02:29             tazmo[0-1]

RESOURCE       CPUS    STATUS        TIME    USERNAME  NODES
parallel.996      2 allocated       00:05        user  tazmo0

JOB            CPUS    STATUS        TIME    USERNAME  NODES
parallel.1115     2   running       00:04        user  tazmo0
```

The machine section gives the name of the machine and the active configuration.

For each partition in the active configuration `rinfo` shows the the number of CPUs in use, the total number of cpus, the time since the partition was started, any CPU time limits imposed on jobs and the node names. This information is extracted from the `partitions` table. The description of the `root` partition shows the resources of the whole machine.

The resource section identifies the resource allocated to the user, the number of CPUs that the resource includes, the node names and the status of the resource. The `time` field specifies how long the resource has been held.

The jobs section identifies the job ID, the number of CPUs the job is using, on which nodes and the status of the job. The `time` field specifies how long the job has been running in hours, minutes and seconds.

EXAMPLES

When used with the `-q` flag `rinfo` will print information on the current user's project codes, resource usage, default memory limit and default priority.

```
duncan@pestilencei: rinfo -q
PARTITION       CLASS         NAME     CPUS   MEMLIMIT   PRIORITY
parallel      project      default      0/8        100          0
parallel      project      divisionA  16/64       none          1
```

In this case access controls allow any user to run jobs on up to 8 CPUs with a memory limit of 100MB. Jobs submitted with the `divisionA` project run at priority 1, have no memory limit and can use up to 64 CPUs. 16 of these 64 CPUs are in use.

When used with the `-s` option `rinfo` prints information on the status of each of the rms servers.

```
duncan@plaguei: rinfo -l -s all
SERVER               HOSTNAME     STATUS         PID
tlogmgr              rmshost      running        239241
eventmgr             rmshost      running        239246
mmanager             rmshost      running        239260
swmgr                rmshost      running        239252
pmanager-parallel    rmshost      running        239175
duncan@plaguei: rinfo -l -s rmsd
SERVER               HOSTNAME     STATUS         PID
rmsd                 plague0      running        740600
rmsd                 plague1      running        1054968
rmsd                 plague2      running        1580438
rmsd                 plague3      running        2143669
rmsd                 plaguei      running        239212
```

In the above example the system is functioning correctly. In the following example one of the nodes has crashed

```
duncan@plaguei: rinfo -l -s rmsd
SERVER               HOSTNAME     STATUS         PID
rmsd                 plague0      running        740600
rmsd                 plague1      running        1054968
rmsd                 plague2      not responding
rmsd                 plague3      running        2143669
rmsd                 plaguei      running        239212
```

NAME

rmsexec – runs a sequential program on a lightly loaded node

SYNOPSIS

rmsexec [-hv] [-p *partition*] [-s *stat*] [*hostname*] *program* [*args ...*]

OPTIONS

-h	Display the list of options.
-v	Specifies verbose operation.
-p *partition*	Specifies the target partition. The request will fail if load balancing is not enabled on the partition.
-s *stat*	Specifies the statistic on which to base the load balancing calculation (see below).

DESCRIPTION

The rmsexec program provides a mechanism for running sequential programs on lightly loaded nodes – nodes, for example, with free memory or low CPU usage. It locates a suitable node and then runs the *program* on it.

The user can select a node from a specific partition (of type login or general) with the -p option. Without the -p option rmsexec uses the default load balancing partition (specified with the lbal-partition attributes in the attributes table). In addition, the *hostname* of the node can be specified explicitly. The request will fail if this node is not available to the user. System administrators may select any node.

The -s option can be used to specify a statistic on which to base the loading calculation. Available statistics are:

usercpu	Percentage of CPU time spent in the user state.
syscpu	Percentage of CPU time spent in the system state - a measure of the I/O load on a node.
idlecpu	Percentage of CPU time spent in the idle state.

`freemem` Free memory in MBytes.

`users` Lowest number of users.

By default, `usercpu` is used as the statistic. Statistics can be used on their own, in which case a node is chosen that is lightly loaded according to this statistic, or you can specify a threshold using *`statistic < value statistic > value`*

EXAMPLES

Some examples follow

```
user@tazmo: rmsexec -s usercpu myprog
user@tazmo: rmsexec -s "usercpu < 50" myprog
user@tazmo: rmsexec -s "freemem > 256" myprog
```

SEE ALSO

`rinfo`

NAME

`rmsquery` – submits SQL querys to the RMS database

SYNOPSIS

`rmsquery` `[-huv] [-d` *`name`*`] [-m` *`machine`*`] [`*`SQLquery`*`]`

OPTIONS

`-d` *`name`*	Select database by name.
`-h`	Display the list of options.
`-m` *`machine`*	Select database by machine name.
`-u`	Print dates as seconds since January 1st 1970. The default is to print dates as a string created with `localtime(3)`.
`-v`	Verbosely prints field names above each column of output.

DESCRIPTION

`rmsquery` is used to submit SQL queries to the RMS database. Users are restricted to using the `select` statement to extract information from the database. System administrators may also submit queries that update the database: `create`, `delete`, `drop`, `insert` and `update`. Note that queries modifying the database are logged.

When used without arguments, `rmsquery` operates interactively and a sequence of commands can be issued.

When used interactively `rmsquery` supports GNU readline and history mechanisms. Type `history` to see recent commands, use `Ctrl/p` and `Ctrl/n` to step back and forward through them. Other builtin commands include `tables` which lists the tables and `fields` followed by the name of a table that lists the fields in a table. The command `verbose` toggles printing of fieldnames.To quit interactive mode, type `Ctrl/d` or `exit` or `quit`.

`rmsquery` is distributed under the terms of the GNU General Public License (see `http://www.gnu.org` for details and more information on GNU readline and history). The source is provided in `/opt/rms/src`.

EXAMPLES

An example of a `select` statement that results in a list of the names of all the nodes in the machine. Note that the *`query`* must be quoted. This is because `rmsquery` expects a single argument.

```
duncan@tazmo: rmsquery "select name from nodes"
tazmo0
tazmo1
tazmo2
tazmo3
```

In the following example, `rmsquery` is used to print information on all jobs run by a user:

```
duncan@tazmo: rmsquery "select name,status,hostnames,ncpus,startTime,endTime from
                        resources where username='duncan'"
7     finished  pestilence[0-3]   4      12/21/99 11:16:44  12/21/99 11:16:46
8     finished  pestilence0       2      12/21/99 11:54:23  12/21/99 11:54:29
9     finished  pestilence[0-3]   4      12/21/99 11:54:35  12/21/99 11:54:39
```

The `-v` option prints field names. In the following example, `rmsquery` it is used to print resource usage statistics:

```
duncan@tazmo: rmsquery -v "select * from acctstats"
name  uid   project  started              etime     atime     utime     stime
-------------------------------------------------------------------------------
7     1507  1        12/21/99 11:16:44      2.00      8.00      0.10      0.22
8     1507  1        12/21/99 11:54:23      6.65     13.30     10.62      0.10
9     1507  1        12/21/99 11:54:35      4.27     16.63     12.28      0.44
```

When used without arguments, `rmsquery` operates interactively and a sequence of commands can be issued.

```
duncan@tazmo: rmsquery -v
sql> select name, status from partitions
name       status
-----------------
login      running
parallel   running
sql>
```

B

Shmem Library Routines

B.1 Overview

This appendix itemizes the Shmem routines, noting which are supported and which are not. The routines are grouped in the following categories:

- Initialization (Section B.1.1)
- Cache (Section B.1.2)
- Accessibility (Section B.1.3)
- Synchronization with put and get (Section B.1.4)
- Put and get (Section B.1.5)
- Strided or indexed put and get (Section B.1.6)
- Collective communications (Section B.1.7)
- Atomic operations (Section B.1.8)
- Remote synchronization (Section B.1.9)
- Remote locking (Section B.1.10)

B.1.1 Initialization Routines

```
start_pes ()    num_pes ()
my_pe ()
```

The three initialization routines are fully supported. `start_pes()` expects all of the processes (also known as processing elements or PEs) to have been started by RMS. The function initializes the caller and then synchronizes the caller with the other processes. The rank of the process is returned by `my_pe()`. The total number of processes is returned by `num_pes()`.

B.1.2 Cache Routines

```
shmem_clear_cache_inv ()      shmem_udcflush ()
shmem_set_cache_inv ()        shmem_udcflush_line ()
shmem_set_cache_line_inv ()
```

The cache routines maintain cache coherency on Cray systems. They are implemented as `NOP`s on the Compaq AlphaServer SC. That is, the routines perform no operation; they just return to the caller successfully.

B.1.3 Access Routines

```
shmem_stack ()    shmem_ptr ()
```

In general, Shmem supports access to a contiguous region of the virtual address space, starting at the base of the DATA segment and extending up past the BSS and the heap of the process. Stack access is not supported for Compaq AlphaServer SC Version 1.0 software; Calling `shmem_stack()` or `shmem_ptr()` will causes a fatal error.

B.1.4 Synchronization Routines

```
shmem_fence ()    shmem_quiet ()
```

The sychronization routines are fully supported. The initial implementation of `shmem_fence()` just calls `shmem_quiet()`. `shmem_quiet()` waits for all outstanding Elan operations to complete. <! [%notfordigital; [See Elan Programming Manual for more information about the Elan.]]

B.1.5 Put and Get Routines

```
shmem_short_g ()          shmem_short_p ()
shmem_int_g ()            shmem_int_p ()
shmem_long_g ()           shmem_long_p ()
shmem_float_g ()          shmem_float_p ()
shmem_double_g ()         shmem_double_p ()

shmem_put ()              shmem_get ()
shmem_put32 ()            shmem_get32 ()
shmem_put64 ()            shmem_get64 ()
shmem_put128 ()           shmem_get128 ()
shmem_putmem ()           shmem_getmem ()

shmem_double_put ()       shmem_double_get ()
shmem_float_put ()        shmem_float_get ()
shmem_int_put ()          shmem_int_get ()
shmem_long_put ()         shmem_long_get ()
shmem_longdouble_put ()   shmem_longdouble_get ()
shmem_longlong_put ()     shmem_longlong_get ()
shmem_short_put ()        shmem_short_get ()
```

The put and get routines are all fully supported.

B.1.6 Strided or Indexed Put and Get Routines

```
shmem_iget ()             shmem_iput ()
shmem_iget32 ()           shmem_iput32 ()
shmem_iget64 ()           shmem_iput64 ()
shmem_iget128 ()          shmem_iput128 ()
shmem_short_iget ()       shmem_short_iput ()
shmem_int_iget ()         shmem_int_iput ()
shmem_long_iget ()        shmem_long_iput ()
shmem_longlong_iget ()    shmem_longlong_iput ()

shmem_float_iget ()       shmem_float_iput ()
shmem_double_iget ()      shmem_double_iput ()
shmem_longdouble_iget ()  shmem_longdoulble_iput ()
```

(continued on next page)

(continued from previous page)

```
shmem_ixget ()              shmem_ixput ()
shmem_ixget32 ()            shmem_ixput32 ()
```

The strided and indexed put and get routines are fully supported. They are implemented as repeated calls to the basic put and get routines listed in Section B.1.5.

B.1.7 Collective Communications Routines

```
barrier ()                          shmem_broadcast ()
shmem_barrier_all ()                shmem_broadcast32 ()
shmem_barrier ()                    shmem_broadcast64 ()

shmem_collect ()                    shmem_fcollect ()
shmem_collect32 ()                  shmem_fcollect32 ()
shmem_collect64 ()                  shmem_fcollect64 ()

shmem_short_sum_to_all ()           shmem_short_prod_to_all ()
shmem_int_sum_to_all ()             shmem_int_prod_to_all ()
shmem_float_sum_to_all ()           shmem_float_prod_to_all ()
shmem_double_sum_to_all ()          shmem_double_prod_to_all ()
shmem_longdouble_sum_to_all ()      shmem_longdouble_prod_to_all ()
shmem_complexf_sum_to_all ()        shmem_complexf_prod__to_all ()
shmem_complexd_sum_to_all ()        shmem_complexd_prod_to_all ()

shmem_short_max_to_all ()           shmem_short_min_to_all ()
shmem_int_max_to_all ()             shmem_int_min_to_all ()
shmem_float_max_to_all ()           shmem_float_min_to_all ()
shmem_double_max_to_all ()          shmem_double_min_to_all ()
shmem_longdouble_max_to_all ()      shmem_longdouble__min_to_all ()

shmem_short_or_to_all ()            shmem_short_and_to_all ()
shmem_int_or_to_all ()              shmem_int_and_to_all ()

shmem_short_xor_to_all ()
shmem_int_xor_to_all ()
```

The collective communcations routines are fully supported, including parameters that specify subsets from the set of processes.

B.1.8 Atomic Routines

```
shmem_short_add ()      shmem_short_inc ()

shmem_short_fadd ()     shmem_short_finc ()

shmem_swap ()
shmem_short_swap ()     shmem_short_mswap ()
shmem_int_swap ()       shmem_int_mswap ()
shmem_long_swap ()      shmem_long_mswap ()
shmem_float_swap ()
shmem_double_swap ()

shmem_short_cswap ()
shmem_int_cswap ()
shmem_long_cswap ()
```

The atomic routines are supported. Atomicity is only guaranteed if the addresses passed are updated solely by the Shmem routines. Atomic operations are performed by an Elan thread on the target node. Applications using the 16 bit atomics on Alpha systems must be compiled with the `-ev6` option.

B.1.9 Remote Synchronization Routines

```
shmem_wait ()    shmem_wait_until ()
```

The remote synchronization routines are supported. They wait until a store location is modified by a put from another node. This means that synchronization is only guaranteed if the addresses passed are updated solely by Shmem routines.

B.1.10 Remote Locking

```
shmem_clear_lock ()    shmem_test_lock ()
shmem_set_lock ()
```

The remote locking routines are not supported for Compaq AlphaServer SC Version 1.0 software. Calling them causes a fatal error.

C

Elan Library Environment Variables

C.1 Using Environment Variables

The Elan library provides a set of tagged message passing routines, which make use of tagged message ports, known as *tport*s, for point-to-point communications. The following environment variables can be used to tune the behaviour; of these routines. Since the MPI library is layered on top of the tagged message passing routines, these environment variables also affect the performance of MPI programs on Compaq AlphaServer SC systems.

`LIBELAN_TPORT_BIGMSG=bytes`

Messages that are larger (in bytes) than the value of `LIBELAN_TPORT_BIGMSG` are sent only when a matching receive has been posted. This means the transfer is synchronous and the receiver can limit the size of receive message buffers. The default value of the variable is 4MBytes.

`LIBELAN_SHM_ENABLE=1`

This variable enables or disables communications within a node being transferred via shared memory. The default value is `TRUE` (`1`).

`LIBELAN_ALLOC_SIZE=bytes`

This variable defines the amount of virtual memory (in bytes) that is allocated for use by the MPI system buffer pool. The default value is 200MBytes.

C.2 Troubleshooting

MPI programs that send large numbers of messages without performing matching receives will eventually run out of system buffer memory. If this happens you will get the message

```
tportBuf: Main memory exhausted ...
```

You can put off (or in some cases avoid) this problem by increasing the size of the buffer pool, but in general you should not rely on system buffering; it uses up memory and reduces performance.

Glossary

Abbreviations

API Application Program Interface — specification of interface to software package (library).

CFS Cluster File System — the remote file system for OSF1 UNIX clusters.

CGI Common Gateway Interface — a standard method for generating HTML pages dynamically from an application so that a Web server and a Web browser can exchange information. A CGI script can be written in any language and can access various types of data, for example, an SQL database.

CPU Central Processing Unit — the part of the computer that executes the machine instructions that make up the various user and system programs.

CRC Cyclic Redundancy Check —

CVS Concurrent Versions System — a revision control utility for managing software releases and controlling the concurrent editing of files by multiple software developers.

DIMM Dual In-Line Memory Module.

DMA Direct Memory Access — high performance I/O technique where peripherals read/write memory directly and not through a CPU.

GNU GNU's Not UNIX — A UNIX-like development effort of the Free Software Foundation, headed by Richard Stallman.

HTML	HyperText Markup Language — a generic markup language, comprising a set of tags, that enables structured documents to be delivered over the WorldWide Web and viewed by a browser.
HTTP	HyperText Transfer Protocol — a communications protocol commonly used between a Web server and a Web browser together with a URL (Uniform Resource Locator).
LED	Light-Emitting Diode;
MIMD	Multiple Instruction, Multiple Data — parallel processing computer architecture characterized as having multiple processors each (potentially) executing a different instruction sequence on different data.
MMU	Memory Management Unit — part of CPU that provides protection between user processes and support for virtual memory.
MPI	Message Passing Interface — high level parallel processing API.
MPP	Massively Parallel Processing — processing that involves the use of a large number of processors in a coordinated fashion.
PCI	Peripheral Component Interconnect — the Elan is connected to a node through this interface.
PDF	Portable Document Format — the page description language used by Adobe Acrobat, derived from PostScript, for displaying pages on the screen.
PTE	Page Table Entry — an entry in the page table which maps the base address of a page to physical memory.
RISC	Reduced Instruction Set Computer — a computer whose machine instructions represent relatively simple operations that can be executed very quickly.
RMS	Resource Management System — Quadrics software.
SDRAM	Synchronous Dynamic Random Access Memory — high performance computer memory architecture.
SMP	Symmetric Multi-Processor — a computer whose main memory is shared by more than one processor.

SQL Structured Query Language — a database language.

TLB Translation Lookaside Buffer — part of the MMU that caches the result of virtual to physical address translations to minimize translation times in subsequent accesses to the same page.

URL Uniform Resource Locator — a standard protocol for addressing information on the World Wide Web.

UTC Coordinated Universal Time[1] — on UNIX® systems it is represented as the time elapsed in seconds since January 1st, 1970 at 00:00:00.

Terms

barrier A synchronisation point in a parallel computation that all of the processes must reach before they are allowed to continue.

bi-sectional bandwidth
The worst case bandwidth across the diameter of the network.

block A thread that blocks relinquishes the processor until a specified event occurs.

critical section A section of program statements that can yield incorrect results if more than one thread tries to execute the section at the same time.

Elan memory The SDRAM on the Elan card.

event A parallel processing synchronisation primitive implemented by the Elan card.

fail-over Swapping from one layer to another in the event of a failure.

Flit A comms cycle unit of information.

HTTP cookies Cookies provide a general mechanism that HTTP server-side connections use to store and to retrieve information on the client side of the connection.

local memory See *Elan memory*

[1] Used to be called GMT

main memory The memory normally associated with the main processor, that is to say, memory on the CPU's high speed memory bus.

main processor The main CPU (or CPUs for a multi-processor) of a node, typically an Alpha 21264.

management network

A private network used by the RMS daemons for control and diagnostics.

multi-rail system

A system that has more than one Elan card connected to each node, each Elan card being connected to a different switch network.

multi-threaded program

A multi-threaded program is one that is constructed such that, during its execution, multiple sequences of instructions are executed concurrently (possibly by different CPUs). Each thread of execution has a separate stack but otherwise they all share the same address space.

node A system with memory, one or more CPUs and one or more Elan cards running an instance of the operating system.

poll Loop and check on each loop whether a specified event has occurred.

rank An integer value that identifies a single process from a set of parallel processes.

reduce Combine the results of a parallel computation into a single value.

remote memory The memory (Elan card or main) of a node when accessed by another node over the network.

resource A set of CPUs allocated to a user to run one or more parallel jobs.

slice A local copy of a global object.

switch network The network constructed from the Elan cards and Elite cards.

thread An independent sequence of execution. Every host process has at least one thread.

virtual memory A feature provided by the operating system, in conjunction with the MMU, that provides each process with a private address space that may be larger than the amount of physical memory accessible to the CPU.

virtual process A (possibly multi-threaded) component of a parallel program executing on a node.

word A 64-bit value.

Index

R

S

T

V